Julia. Projects from a Recreational Programmer’s Drawer

Bartlomiej Lukaszuk

Version: 2026-04-29

Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International

Contents

About 1
Matrix Multiplication 4
Bat and Ball 10
Camel Case 14
Toothpaste 17
Bile 22
Leap Year 28
Calendar 32
Prime Numbers 42
Recursion 46
Randomness 52
Birthday 62
Logo 68
Binary 74
The Answer 84
The Doors 88
Progress Bar 96
Pascal's Triangle 102
Lattice Paths 110
Stem and Leaf Plot 120
Canvas 128
Tree 136
Format Text 142
Roman Numerals 151
Cheque 157
Sort 163
Regex 170
Molar Mass 191
Tic-Tac-Toe 204
Touch Typing 214
Compound Interest 221
Mortgage 233
Overpayment 242

Diffusion 250
Transcription 259
Translation 264
Altruism 271
Shift 279
Caesar 284
Vigenere 290
Game of Life 297
The end 303

About

Hi, I'm Bart and this is my second open access book entitled:

"Julia. Projects from a Recreational Programmer's Drawer"

> ***Note:*** The book was written by an amateur programmer (with all its potential negative consequences). If you find this a problem, stop reading now.

The book contains a set of challenges of varying level of complexity (most likely in the range of easy to moderate). The exercises are for the problems that, for whatever reason, I found interesting and/or suitable. The tasks are accompanied by exemplary solutions in Julia[1] (with explanations). Still, I recommend you try to solve the tasks yourself. Alternatively you may read the solutions and try to recreate them as much as you can on your own. The key thing is, if you want to learn, write the code.

For practical reasons, I will assume this book is read by curious readers of non-mathematical (i.e. resembling mine) background. Moreover, I expect that the readers have already mastered the language basics and now are on a lookout for a way to hone their newly acquired skills. To that end, I'll imagine you have read my previous open access book[2]. I'll do this not not because it is the best book in the world (which it is), but because of the DRY principle[3](I'm going to apply similar conventions without delving too much into the previously mentioned topics).

For instance, just like in the previous book, here I will use the assert[4] macro to test a function's assumptions and print error messages. The construct is not recommended in a serious program (see the warning in the docs), but for the purpose of this book it should do the trick.

---

[1]https://julialang.org/

[2]https://b-lukaszuk.github.io/RJ_BS_eng/

[3]https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

[4]https://docs.julialang.org/en/v1/base/base/#Base.@assert

Additionally, henceforth I will define a few type aliases, like:

```
const Flt = Float64
const Str = String
const Vec = Vector
```

This will allow for a shorter code when type declarations are used, e.g. `Vec{Flt}` instead of `Vector{Float64}`. Notice, that the type synonyms are declared with `const` keyword, since they will not change for as long as a program runs. The naming convention for the custom types is similar to the name of the built in data types in Julia (first letter is uppercased, the rest of the characters are lowercased).

I wrote this book using Julia:

```
VERSION
```

1.10.11

so an LTS[5] edition with the following internal (available to you after installment):

- Base
- Dates
- Random
- Statistics
- Test

and external (downloaded from the internet separately) libraries:

- BenchmarkTools
- CairoMakie
- DataFrames
- Makie
- Symbolics

Still, you should be able to solve the tasks with the functionality that comes with your installation (not necessarily the ones named above).

[5] https://en.wikipedia.org/wiki/Long-term_support

If, for any reason, this book is not to your taste then feel free to visit, e.g. Adam Wysokinski's the Big Book of Julia[6]and choose a learning resource of your liking. Alternatively you may visit Rosetta Code webpage[7]that contains over 1'000 programming exercises with solutions in different programming languages. Chances are that many of the exercises presented here are to be found there (not that I copied them, it's just that they've been around for quite some time and I don't even know whom should I give credit for them).

Finally, just like in the previous book, I'll try to write in a possibly simple (clarity over cleverness and performance) and correct manner. Still, I'm only human, so watch out for possible errors and bugs. Anyway, I hope the book will satisfy your appetite, it is available freely under Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International[8]license.

Let the games begin.

[6]https://adamwysokinski.codeberg.page/bbj/
[7]https://rosettacode.org/wiki/Category:Solutions_by_Programming_Task
[8]http://creativecommons.org/licenses/by-nc-sa/4.0/

Matrix Multiplication

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[9](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[10].

Problem

In Julia a matrix[11]is a tabular representation of two-dimensional (rows and columns) data. We declare it with a friendly syntax (columns separated by spaces, rows separated by semicolons).

```
A = [10.5 9.5; 8.5 7.5; 6.5 5.5]
A
```

```
3×2 Matrix{Float64}:
 10.5  9.5
  8.5  7.5
  6.5  5.5
```

In mathematics by convention we denote matrices with a single capital letter. However, since I'm not a mathematician then I'll use the lowercase names (they are easier for the fingers).

I remember that the first time that I was introduced to the matrix algebra I found it pretty boring and burdensome. However, believe it or not, matrices are very useful in mathematics and in everyday life, e.g statistical programs or programs rendering computer graphics rely

[9]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/matrix_multiplication

[10]https://docs.julialang.org/en/v1/stdlib/Pkg/

[11]https://b-lukaszuk.github.io/RJ_BS_eng/julia_language_variables.html#sec:julia_arrays

on them heavily (chances are you used them without even being aware of it).

Anyway, here is the task. Read about matrix multiplication, e.g. on Math is Fun[12]or watch a Khan Academy's video[13]on the topic and write a function with the following signature

```
multiply(m1::Matrix{Int}, m2::Matrix{Int})::Int
```

that for

```
# Math is Fun example
a = [1 2 3; 4 5 6]
```

```
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

and

```
# Math is Fun example
b = [7 8; 9 10; 11 12]
```

```
3×2 Matrix{Int64}:
  7   8
  9  10
 11  12
```

Should return the following matrix

```
[58 64; 139 154]
```

```
2×2 Matrix{Int64}:
  58   64
 139  154
```

[12]https://www.mathsisfun.com/algebra/matrix-multiplying.html
[13]https://www.youtube.com/watch?v=OMA2Mwo0aZg

Compare `multiply` against Julia's built-in `*` operator (on some matrices of your choice) to ensure its correct functioning.

Solution

It appears that in order to multiply two matrices we need to multiply each element in a row from a matrix by each element in a column of another matrix. Once we are done we sum the products. This is called a dot product. So, let's start with that.

```
function getDotProduct(row::Vec{Int}, col::Vec{Int})
    @assert length(row) == length(col) "row & col must be of equal
length"
    return map(*, row, col) |> sum
end
```

> Note. Thanks to the previously defined (Section 1) type synonyms we saved some typing and used `Vec{Int}` instead of `Vector{Int}`. We will use such small convenience(s) throughout the book. The type synonyms are defined in Section 1 and/or in the code snippets for each chapter.

First, we place a simple assumption check with the assert[14]. Then we multiply each element of `row` by each element of `col` with `map`. `Map` applies a function (its first argument) to every element of a collection (its second argument), like so:

```
# adds 10 to each element of a vector
map(x -> x + 10, [1, 2, 3])
```

```
[11, 12, 13]
```

Here we used a vector (`[1, 2, 3]`) and applied an anonymous function (`x -> x + 10`) to each of its elements. The function accepts one argument (`x`), adds 10 to it (`x + 10`) and returns (`->`) that value. Since `x` will become every element of the vector `[1, 2, 3]` then in

[14]https://docs.julialang.org/en/v1/base/base/#Base.@assert

effect 10 will be added to the each component of the vector and the results will be collected into a new vector (the old vector is not changed). Interestingly, we may also use a function that accepts two arguments and apply this function to parallel elements of two vectors, like so:

```
map((x, y) -> x * y, [1, 2, 3], [10, 100, 1000])
```

```
[10, 200, 3000]
```

Here, `x` becomes every value of `[1, 2, 3]` and `y` every value of `[10, 100, 1000]` vector. Given that * is just a syntactic sugar for `*(x, y)` we may simply place * alone.

```
map(*, [1, 2, 3], [10, 100, 1000])
```

```
[10, 200, 3000]
```

Since we calculate a dot product, then as an alternative (to live up to its name) we could also use the dot operator[15]syntax in our `getDotProduct` function like so:

```
[1, 2, 3] .* [10, 100, 1000]
```

```
[10, 200, 3000]
```

Anyway, once we got the vector of products we send it (|>[16]) as an input to `sum`.

OK, time for the multiplication itself.

```
function multiply(m1::Matrix{Int}, m2::Matrix{Int})::Matrix{Int}
    nRowsMat1, nColsMat1 = size(m1)
```

[15]https://b-lukaszuk.github.io/RJ_BS_eng/julia_language_repetition.html#sec:julia_language_dot_functions

[16]https://docs.julialang.org/en/v1/base/base/#Base.:%7C%3E

```
    nRowsMat2, nColsMat2 = size(m2)
    @assert  nColsMat1 == nRowsMat2 "the matrices are incompatible"
    result::Matrix{Int} = zeros(nRowsMat1, nColsMat2)
    for r in 1:nRowsMat1
        for c in 1:nColsMat2
            result[r, c] = getDotProduct(m1[r,:], m2[:, c])
        end
    end
    return result
end
```

The above is a translation of the algorithm from the links provided in the task description (see Section 2.1). First we get our matrices dimensions and perform a compatibility check with `@assert`. Then we initialize an empty matrix (`result`) with the appropriate dimensions (we use `zeros`, so 0s are the placeholders stored in its cells). Finally, we get the dot products of every row (`for r`) in `m1` by every column (`for c`) in `m2` and place them to the appropriate cells in the `result` matrix.

Alternatively, if you are not a fan of nesting, you may use Julia's simplified nested for loop syntax. It works the same as the previous code snippet.

```
function multiply(m1::Matrix{Int}, m2::Matrix{Int})::Matrix{Int}
    nRowsMat1, nColsMat1 = size(m1)
    nRowsMat2, nColsMat2 = size(m2)
    @assert  nColsMat1 == nRowsMat2 "the matrices are incompatible"
    result::Matrix{Int} = zeros(nRowsMat1, nColsMat2)
    for r in 1:nRowsMat1, c in 1:nColsMat2
        result[r, c] = getDotProduct(m1[r,:], m2[:, c])
    end
    return result
end
```

Anyway, let's give it a swing.

```
# Math is Fun examples
a = [1 2 3; 4 5 6]
b = [7 8; 9 10; 11 12]
multiply(a, b)
```

```
2×2 Matrix{Int64}:
  58   64
 139  154
```

Looks good, and

```
# Khan Academy examples
c = [-1 3 5; 5 5 2]
d = [3 4; 3 -2; 4 -2]
multiply(c, d)
```

```
2×2 Matrix{Int64}:
 26  -20
 38    6
```

Appears to be correct as well.

And now for a few tests against the build in * operator.

```
(a * b) == multiply(a, b)
(c * d) == multiply(c, d)
```

true

We can't complain. It appears that we managed to solve this task in like 15 lines of code and without over-engineering it too much. It's all thanks to the Julia's nice and terse syntax.

Bat and Ball

In this chapter I used the following libraries.

```
import Symbolics as Sym # external library
```

Still, once you read the problem description you may decide to do otherwise.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[17](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[18].

Problem

Recently, someone told me an interesting small mathematical problem that I happened to knew from my youth:

> A bat and a ball cost in total $1.1. The bat costs $1 more than the ball. How much costs the ball?

Pause for a moment and try to find the answer. If you are stuck, then do this with pen and paper first. Then watch this[19]video and look for Julia's built in functionality that will speed up the longhand calculations.

Solution

My first impulse was that the ball should cost $0.1 or 10 cents. It's the only logical solution, right?

Surprisingly, it turns out that this simple problem trips up a lot of people (if you were not one of them, congrats!).

---

[17]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/bat_and_ball

[18]https://docs.julialang.org/en/v1/stdlib/Pkg/

[19]https://www.youtube.com/watch?v=AUqeb9Z3y3k

Let's summon primary school math to the rescue and settle this once and for all. From the task description we know that:

$$bat + ball = 1.1 \qquad (1)$$

$$bat - ball = 1 \qquad (2)$$

Therefore, we can rewrite the Equation 2 (move `- ball` to the other side and change mathematical operation to the opposite) to get:

$$bat = 1 + ball \qquad (3)$$

Finally, by substituting `bat` from Equation 1 with `bat` from Equation 3 (`bat = 1 + ball`) we get.

$$1 + ball + ball = 1.1$$

which we can simplify to

$$1 + 2*ball = 1.1$$

$$2*ball + 1 = 1.1$$

$$2*ball = 1.1 - 1$$

$$2*ball = 0.1$$

$$ball = 0.1/2$$

to finally get:

$$ball = 0.05$$

So it turns out that, counter-intuitively, the ball costs $0.05 or 5 cents.

That's all very interesting, but what any of this got to do with Julia? Well, just that we can solve this and more complicated equations with our favorite programming language. For that purpose we will use matrices and their multiplications as explained in this[20]Khan Academy's video.

[20]https://www.youtube.com/watch?v=AUqeb9Z3y3k

```
variables = [
    1 1; # 1 bat + 1 ball
    1 -1 # 1 bat - 1 ball
]
```

```
2×2 Matrix{Int64}:
 1   1
 1  -1
```

First we set the `variables` matrix where row 1 represents the left side of Equation 1 and row 2 stands for the left side of Equation 2 . Column 1 contains the number of `bats` in each equation, whereas column 2 the number of `balls`.

And now for the right sides of the equations, we will place them in the `prices` vector.

```
prices = [1.1, 1.0]
```

```
[1.1, 1.0]
```

All that's left to do, is to multiply the inverse (`inv`) of the matrix `variables` by the `prices`.

```
result = inv(variables) * prices
# or, shortcut
result = variables \ prices
round.(result, digits=4)
```

```
[1.05, 0.05]
```

Here we see the prices of `bat` (1.05) and `ball` (0.05) calculated by Julia. We rounded the results to counterbalance inexact float representation in computers (as discussed previously[21]).

[21]https://b-lukaszuk.github.io/RJ_BS_eng/julia_language_variables.html#sec:julia_float_comparisons

Now, if you're new to matrix algebra, then this may look like an unnecessary hassle and some obscure alchemy. In that case you may consider using Symbolics.jl[22], a package with a bit friendlier and more human readable syntax.

```
import Symbolics as Sym

Sym.@variables bat ball
result = Sym.symbolic_linear_solve(
    [
        bat + ball ~ 1.1,
        bat - ball ~ 1
    ],
    [bat, ball]
);
map(x -> round(x.val, digits=4), result)
```

```
[1.05, 0.05]
```

First, we declare variables (`Sym.@variables`) that we will use in our equations (customarily those are `x`, `y`, `z`, etc.), here we opted for more human readable `bat` and `ball` names. Next we use `symbolic_linear_solve` function to get the solution (the calculation process may take a second or two). It takes 2 arguments (separated by coma): 1) equation(s) and 2) variable(s) for which we want to solve our equation. Since we got a set of 2 equations we place them in square brackets separated by comma. Inside the equations we use previously defined (`Sym.@variables`) variables (`bat` and `ball`) and `~` instead of `=` known from mathematics. Next, we send the variable(s) we are looking for (`[bat, ball]`). The number of variables should be equal to the number of equations in the first argument, and if it is greater than 1 then we place them between square braces and separate them with comas. And that's it.

Pretty neat trick. Worth to know if your math is rusty (like mine) and you want to confirm your pen and paper calculations.

[22] https://github.com/JuliaSymbolics/Symbolics.jl

Camel Case

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[23](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[24].

Problem

In programming there are a few different types of naming conventions, the two most popular of them are: smallCamelCase[25]and snake_case[26].

At times it is useful to quickly convert from one of them to the other (hence such a functionality is sometimes found in IDEs[27]).

Here is a task for you. Write two functions with the following signatures:

```
changeToCamelCase(snakeCasedWord::Str)::Str
```

and

```
changeToSnakeCase(camelCasedWord::Str)::Str
```

The functions should perform the conversion as specified in this template:

[23]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/camel_case

[24]https://docs.julialang.org/en/v1/stdlib/Pkg/

[25]https://en.wikipedia.org/wiki/Camel_case

[26]https://en.wikipedia.org/wiki/Snake_case

[27]https://en.wikipedia.org/wiki/Integrated_development_environment

```
"hello_world" <=> "helloWorld"
"nice_to_meet_you" <=> "niceToMeetYou"
"translate_to_english" <=> "translateToEnglish"
```

You may assume that the input is well formatted and contains only the underscores (“_“) and the characters from the Latin alphabet.

Solution

One of the most succinct (and quite performant) solutions would be based on regular expressions[28](also called regexes). Julia does have a regex support (see the docs[29]) and we will explore this venue in Section 27.1 . For now, in order to keep things simple our approach will rely on good old for loops and conditionals.

First `changeToSnakeCase` as it is simpler to write (start small and build).

```
function changeToSnakeCase(camelCasedWord::Str)::Str
    result::Str = ""
    for c in camelCasedWord
        result *= isuppercase(c) ? '_' * lowercase(c) : c
    end
    return result
end
```

We begin with an empty `result`. Next, we travel through each character (`c`) of our `camelCasedWord` if a letter is uppercased (`isuppercase(c) ?`) we update our result (`*=`) by appending to it underscore (`'_'`) concatenated (`*`) with the lowercased character (`lowercase(c)`). Otherwise (`:`) we append the character unchanged (`c`). Finally, we `return` the `result`.

Let’s see if it works.

```
map(changeToSnakeCase, ["helloWorld",
    "niceToMeetYou", "translateToEnglish"])
```

[28]https://en.wikipedia.org/wiki/Regular_expression
[29]https://docs.julialang.org/en/v1/base/strings/#Base.Regex

```
["hello_world", "nice_to_meet_you", "translate_to_english"]
```

Looks good. Time for the other function.

```
function changeToCamelCase(snakeCasedWord::Str)::Str
    result::Str = ""
    prevUnderscore::Bool = false
    for c in snakeCasedWord
        if c == '_'
            prevUnderscore = true
            continue
        else
            result *= prevUnderscore ? uppercase(c) : c
            prevUnderscore = false
        end
    end
    return result
end
```

One more time, we begin with an empty result (`result::Str = ""`), but this time we also declare an indicator that tells us whether the previously examined letter was an underscore (`prevUnderscore`). Next, we traverse the `snakeCasedWord` character by character (`for c in snakeCasedWord`) and build up the result. If the currently examined character is an underscore (`if c == '_'`) we set the indicator to true and skip rest of the code in the for loop (in this iteration only) with continue[30]. Otherwise (`else`), we append the character to the result (`result *=`) with the proper casing based on the value of `prevUnderscore` and set this last variable to `false`. Once we're done, we `return` the `result`.

Time for another test.

```
map(changeToCamelCase,
    ["hello_world", "nice_to_meet_you", "translate_to_english"])
```

```
["helloWorld", "niceToMeetYou", "translateToEnglish"]
```

And another small victory.

[30]https://docs.julialang.org/en/v1/base/base/#continue

Toothpaste

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[31](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[32].

Problem

There is a story (perhaps it's just an urban legend) that in the the 1960s a toothpaste manufacturer faced a financial crisis. They were desperate to make the sales go up, but nothing seemed to work. Finally, a guy came by and offered to increase their sales by at least 50% in exchange for $100,000. At first the company declined the proposal on account of being to expensive and coming from a person with no track record. However, after a year the management saw no other option, but to accept the offer. After all the legal details were set, the guy spoke only one sentence: "Make the hole bigger".

Try to figure out does making the hole bigger actually moves the sales up by $\geq$ 50%. Test different scenarios, e.g. different initial hole size, and see what would have happened if the customers tried to counteract this idea by squeezing less toothpaste (shorter strip) on the toothbrush.

> ***Note:*** $100,000 may not sound like a tone of money today, but if you update it for an inflation rate of let's say 3.6% you will get roughly $1,000,000. Example calculations: 100,000*(1.036^65) ≈ 996,000,000. In that case 65 is the number of years between 1960 and 2025, whereas 1.036 is how much more money you must

[31]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/toothpaste

[32]https://docs.julialang.org/en/v1/stdlib/Pkg/

spend every year on the same product due to the assumed inflation). We will deal with similar calculations in Section 31 .

Solution

First, let me define some abbreviations to use in the code later on.

```
const Vec = Vector
const Flt = Float64
```

Now, based on my own observations I would say that a cylinder[33]is a good approximation of the toothpaste that is squezed out of the tube. Time to define some helper functions that will help us to evaluate the amount of a toothpaste on a toothbrush.

```
struct Cylinder
    radius::Int
    height::Int
end

function getVolume(c::Cylinder)::Flt
    return c.height * pi * c.radius^2
end
```

We will skip the detailed explanations, as the above are just translations of the Wikipedia's formulas into Julia's code.

There is a slight problem with the code above as one could argue that a cylinder mustn't have any of its dimensions equal to or smaller than 0 (`Int` allows for such numbers). So we can either use an unsigned type[34](like `UInt64`) or add a sanity check within a so called inner constructor[35].

```
struct Cylinder
    radius::Int
    height::Int
```

[33]https://en.wikipedia.org/wiki/Cylinder

[34]https://docs.julialang.org/en/v1/manual/integers-and-floating-point-numbers/

[35]https://docs.julialang.org/en/v1/manual/constructors/#man-inner-constructor-methods

```
    Cylinder(r::Int, h::Int) = (r < 1 || h < 1) ?
        error("both radius and height must be >= 1") :
        new(r, h)
end
```

Anyway, in order to answer how much the sales will increase we need to estimate the increase in our toothpaste's usage when the radius and height of our cylinder changes.

```
function getRatios(cylinders::Vec{Cylinder},
                   radiusChange::Int, heightChange::Int)::Vec{Flt}
    ratios::Vec{Flt} = []
    for cyl1 in cylinders
        # newRadius, newHeight, cyl2, vol1, vol2 - local variables
        # visible only in the for loop
        newRadius = cyl1.radius + radiusChange
        newHeight = cyl1.height + heightChange
        cyl2 = Cylinder(newRadius, newHeight)
        vol1 = getVolume(cyl1)
        vol2 = getVolume(cyl2)
        push!(ratios, round(vol2/vol1, digits=2))
    end
    return ratios
end
```

To that end we defined the `getRatios` function that accepts a vector of `cylinders` and the number by which we change their radii (`radiusChange`) and heights (`heightChange`). Next, `for` each cylinder (`cyl1`) in the initial `cylinders` we create its counterpart `cyl2` with the applied size changes. Then we obtain the volumes (`vol1` and `vol2`) for the two cylinders. All that's left to do is to `push` the volume ratio (`vol2/vol1`) into the vector of results (`ratios`) and to `return` it from the function.

Time to test some scenarios.

Scenario 1

We increase the radius of the hole. We assume the change is so small that the customers wouldn't notice (the height remains constant).

```
# radius and height in millimeters
# radius+1, height = does not change
getRatios(Cylinder.(1:5, 5), 1, 0) .* 100
```

```
[400.0, 225.0, 178.0, 156.0, 144.0]
```

> ***Note:*** `Cylinder.(1:5, 5)` creates a vector of `Cylinders` with different radii (1:5) and the same height (5). Moreover, we multiplied (`.*`) the obtained vector of ratios by 100 in order to obtain the change expressed in %.

The data demonstrates that the smaller the initial hole the greater the increase in our toothpaste consumption and therefore the expected sales growth (here in the range of +300% to +44%).

Scenario 2

We increase the radius of the hole, and observe what happens with the consumption of our product if the customers try to squeeze less (shorter strip) of our toothpaste by the very same amount that we increased our radius.

```
# radius and height in millimeters
# radius+1, height-1
getRatios(Cylinder.(1:5, 5), 1, -1) .* 100
```

```
[320.0, 180.0, 142.0, 125.0, 114.99999999999999]
```

Again, we observe a significant growth (although smaller than in the previous scenario) in toothpaste consumption and expected sales (here in the range of +220% to +15%).

Scenario 3

We increase the radius of the hole, and observe what happens with the consumption of our product if the customers try to squeeze less of our toothpaste (they decrease the length of the strip by two units, when we changed the radius by one unit).

```
# radius and height in millimeters
# radius+1, height-2
getRatios(Cylinder.(1:5, 5), 1, -2) .* 100
```

```
[240.0, 135.0, 107.0, 94.0, 86.0]
```

This time the result is inconclusive, because in the two last cases the customers actually use less of our product, and therefore the sales are expected to drop.

Conclusions

In summary, we see that increasing the radius of the hole in a toothpaste is an effective strategy to increase the sales of our product. However, we should restrain ourselves and not overdo it, since if the customers notice they could squeeze shorter toothpaste strips which may undermine our efforts (or even backfire on us).

Bile

In this chapter I used the following libraries.

```
import CairoMakie as Cmk # external library
import Symbolics as Sym # external library
```

Still, once you read the problem description you may decide to do otherwise.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[36](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[37].

Problem

There are different types of nutrients to be found in food, but they can be roughly divided into: sugars (carbohydrates), proteins and fats (lipids). Your liver produces bile[38]that is stored in the gallbladder and released to the duodenum (part of the small intestine). As far as I remember my biology classes bile facilitates digestion by breaking large lipid (fat) droplets into smaller ones. Thanks to that it increases the total surface area in contact with digestive enzymes (lipases). So much for the theory, but I always wondered if that is true.

Use Julia to demonstrate that the total surface area of a few small lipid droplets is actually greater than the surface area of a one big droplet. Of course, the big droplet and small droplets should contain the same volume of lipids.

> ***Hint:*** You will have to assume the shape of a lipid droplet. If your math is rusty, then choose something simple, like a cube[39]. For

[36]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/bile
[37]https://docs.julialang.org/en/v1/stdlib/Pkg/
[38]https://en.wikipedia.org/wiki/Bile

instance, as a starting point you may think about the classical Rubik's cube[40]and into how many pieces you could break it. Then continue by calculating the areas and volumes of the big cube and the small cubes.

Solution

To me the shape that resembles the droplet the most is sphere[41]. Luckily, it also got well defined formulas for surface area and volume (see the link above). So this is what we will use in our solution.

```
struct Sphere
    radius::Flt
    Sphere(r::Flt) = (r <= 0) ? error("radius must be > 0") : new(r)
end

# formula from Wikipedia
function getVolume(s::Sphere)::Flt
    return (4/3) * pi * s.radius^3
end

# formula from Wikipedia
function getSurfaceArea(s::Sphere)::Flt
    return 4 * pi * s.radius^2
end
```

Now, let's define a big lipid droplet with a radius of, let's say, 10 [μm].

```
referenceDroplet = Sphere(10.0)
```

In the next step we will split this big droplet into a few smaller ones of equal sizes. Splitting the volume is easy, we just divide it by the number of droplets. However, we need a way to determine the size (`radius`) of each small droplet. Let's try to transform the formula for a sphere's volume and see if we can get a radius from that.

$$v = \frac{4}{3} * \pi * r^3 \qquad (4)$$

[39]https://en.wikipedia.org/wiki/Cube
[40]https://en.wikipedia.org/wiki/Rubik%27s_Cube
[41]https://en.wikipedia.org/wiki/Sphere

If a = b, then b = a, so we may swap the sides.

$$\frac{4}{3} * \pi * r^3 = v \qquad (5)$$

The multiplication is commutative (the order does not matter), i.e. 2 * 3 * 4 is the same as 4 * 3 * 2 or 2 * 4 * 3, therefore we can rearrange elements on the left side of Equation 5 to:

$$R^3 * \frac{4}{3} * \pi = v \qquad (6)$$

Now, one by one we can move $*\frac{4}{3}$ and $*\pi$ to the right side of Equation 6 . Of course, we change the mathematical operation to the opposite (division instead of multiplication) and get:

$$r^3 = v / \frac{4}{3} / \pi \qquad (7)$$

All that's left to do is to move exponentiation (x^3) to the right side of Equation 7 while changing it to the opposite mathematical operation (cube root, i.e. $\sqrt[3]{x}$).

$$r = \sqrt[3]{v / \frac{4}{3} / \pi} \qquad (8)$$

Now, you might wanted to quickly verify the solution using `Symbolic.symbolic_linear_solve` we met in Section 3.2 . Unfortunately, we cannot use `r^3` (`r` to the 3rd power) as an argument (and solve for `r`), since then it wouldn't be a linear equation (to be linear the power must be equal to 1) required by `_linear_solve`. We could use other, more complicated solver, but instead we will keep things simple and apply a little trick:

```
import Symbolics as Sym

# fraction - 4/3, p - π, r3 - r^3, v - volume
Sym.@variables fraction p r3 v
Sym.symbolic_linear_solve(fraction * p * r3 ~ v, r3) # may take a moment
```

v / (fraction*p)

So, instead of writing the formula as it is, we just named our variables `fraction`, `p`, `r3` and `v`. Anyway, according to `Sym.symbolic_linear_solve` $r^3 = v/\left(\frac{4}{3}*\pi\right)$, which is actually the same as Equation 7 above [since e.g. 18 / 2 / 3 == 18 / (2 * 3)]. Ergo, we may be fairly certain we correctly solved Equation 7 and therefore Equation 8 .

Once, we confirmed the validity of the formula in Equation 8 , all that's left to do is to translate it into Julia code.

```
function getSphere(volume::Flt)::Sphere
    # cbrt - fn that calculates cube root of a number
    radius::Flt = cbrt(volume / (4/3) / pi)
    return Sphere(radius)
end
```

> ***Note:*** If there were no function for $\sqrt[3]{x}$ you could easily define it yourself with: `getCbrt(x) = x^(1/3)` (here I used a single expression function[42]for brevity) since $\sqrt[n]{x} = x^{1/n}$.

Once we got that figured out, we evaluate the total area of the droplets of different size.

```
nDroplets = [1, 4, 8, 12]
totalVolumes = repeat([getVolume(referenceDroplet)], length(nDroplets))
individualVolumes = totalVolumes ./ nDroplets
droplets = getSphere.(individualVolumes)
radii = map(s -> s.radius, droplets)
individualSurfaceAreas = getSurfaceArea.(droplets)
totalSurfaceAreas = individualSurfaceAreas .* nDroplets
```

This seemed like a breeze thanks to the dot operators[43]. We begin by defining the `nDroplets`, i.e. the number of droplets that we will consider. Next, we divide their `totalVolumes` by their numbers (`nDroplets`) to get a volume of an individual droplet

[42] https://en.wikibooks.org/wiki/Introducing_Julia/Functions#Single_expression_functions

[43] https://b-lukaszuk.github.io/RJ_BS_eng/julia_language_repetition.html#sec:julia_language_dot_functions

(`individualVolumes`). Based on the individual volumes we create the `droplets` with the appropriate radii (`getSphere.(individualVolumes)`). We also extract the `radii` for future use (the drawing function below). Now, we calculate `individualSurfaceAreas` of our small droplets (`getSurfaceArea.(droplets)`) and then their `totalSurfaceAreas`. We were able to achieve all that in only 7 lines of code.

Anyway, now, we can either examine the vectors (`totalSurfaceAreas`, `radii`, `nDroplets`, `individualVolumes`) one by one, or do one better and present them on a graph with e.g. CairoMakie (I'm not going to explain the code below, for reference see my previous book[44]or CairoMakie tutorial[45]).

```
import CairoMakie as Cmk

fig = Cmk.Figure();
ax = Cmk.Axis(fig[1, 1],
              title="Lipid droplet size vs. summaric surface area",
              xlabel="number of lipid droplets",
              ylabel="total surface area [μm²]", xticks=0:13);
Cmk.scatter!(ax, nDroplets, totalSurfaceAreas,
             markersize=radii .* 5, color="gold1");
Cmk.xlims!(ax, -3, 16);
Cmk.ylims!(ax, 800, 3000);
Cmk.text!(ax, nDroplets, totalSurfaceAreas .- 150,
    text=map(r -> "single droplet radius = $(round(r, digits=2)) [μm]",
             radii),
    fontsize=12, align=(:center, :center)
);
Cmk.text!(ax, nDroplets, totalSurfaceAreas .- 250,
          text=map((v, n) ->
              "volume ($n droplet/s) = $(round(Int, v)) [μm³]",
                   totalVolumes, nDroplets),
    fontsize=12, align=(:center, :center)
);
fig
```

Behold.

[44]https://b-lukaszuk.github.io/RJ_BS_eng/

[45]https://docs.makie.org/stable/tutorials/getting-started

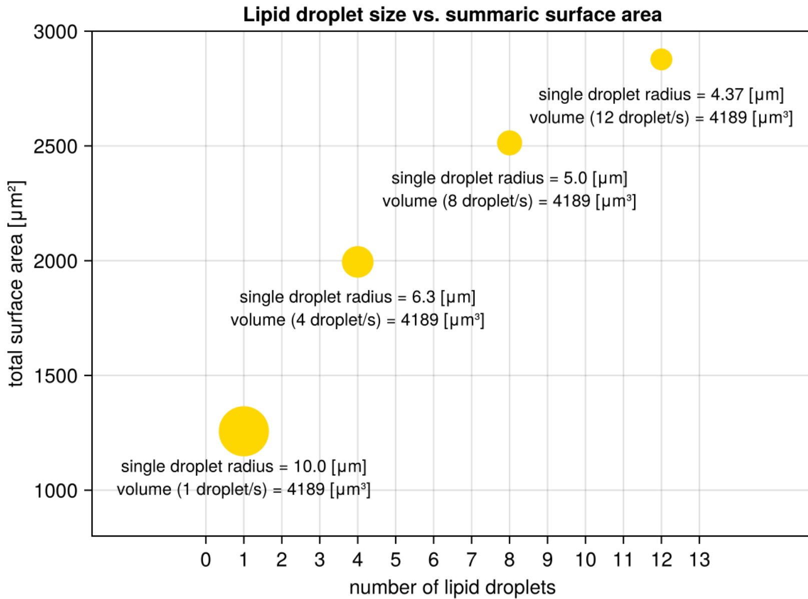


Figure 1: Bile. Splitting a big lipid droplet into a few smaller ones and the effect it has on their total surface area.

So it turns out that what they taught me in the school all those years ago is actually true. But only now I can finally see it. Nice.

> ***Note:*** The above was an example of a geometrical property called surface-to-volume-ratio[46]that applies to more than just the spheres and got implications in many fields of science.

[46]https://en.wikipedia.org/wiki/Surface-area-to-volume_ratio

Leap Year

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[47](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[48].

Problem

As you probably know an astronomical year is slightly more than 365 days (roughly 365.2425 days). For that reason the Gregorian calendar[49] got common years (365 days each) and leap years[50](366 days each).

You task is to write a function that detects whether a given year is a leap year (according to the Gregorian calendar). Feel free to test it, e.g. on the following input: [1792, 1859, 1900, 1918, 1974, 1985, 2000, 2012] of which only 1792, 2000, and 2012 are leap years.

Solution

Let's start with the algorithm, as stated in the Wikipedia's page[51]:

> The rule for leap years is that every year divisible by four is a leap year, except for years that are divisible by 100, except in turn for years also divisible by 400.

Nothing more, nothing less. Time to translate it into Julia's code.

[47]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/leap_year
[48]https://docs.julialang.org/en/v1/stdlib/Pkg/
[49]https://en.wikipedia.org/wiki/Gregorian_calendar
[50]https://en.wikipedia.org/wiki/Leap_year
[51]https://en.wikipedia.org/wiki/Gregorian_calendar

```
function isLeap(yr::Int)::Bool
    @assert 0 < yr < 4001 "yr must be in range [1-4000]"
    divisibleBy4::Bool = yr % 4 == 0
    gregorianException::Bool = (yr % 100 == 0) && (yr % 400 != 0)
    if !divisibleBy4
        return false
    else
        return !gregorianException
    end
end
```

The powerhouse of our function is `%` (modulo operator) that returns the reminder of the division. If a number `x` is evenly divided by a number `y` (`x % y`) the reminder is equal to zero (`== 0`). Otherwise (when `x % y != 0`) the `x` cannot be evenly divided by `y`. Although not strictly necessary, we added a mnemonic names for the tested conditions (`divisibleBy4` and `gregorianException`), which should make the code more readable. Anyway, if a year (`yr`) is not divisible by 4 (`!divisibleBy4`) then it is a common year. Otherwise (`else`) if the the year fulfills the exception rule (`gregorialException` is `true`) it is not a leap year (hence we negate `! gregorianException`, because `! true` is `false`). If a year (`yr`) does not meet the exception criteria (`gregorianException` is `false`) it is a common year (we negate `false`, so we get `true`).

Actually, we can get rid of the `if-else` condition by using && (logical and) and its short circuiting property.

```
function isLeap(yr::Int)::Bool
    @assert 0 < yr < 4001 "yr must be in range [1-4000]"
    divisibleBy4::Bool = yr % 4 == 0
    gregorianException::Bool = (yr % 100 == 0) && (yr % 400 != 0)
    return divisibleBy4 && !gregorianException
end
```

When `divisibleBy4` is `false` the and operator (&&) skips the evaluation of its second argument (short circuiting) and returns `false`. Otherwise, (`divisibleBy4` is `true`) `!gregorianException` is evaluated and it becomes the result of the function.

Time for a simple test.

```
yrs = [1792, 1859, 1900, 1918, 1974, 1985, 2000, 2012]
filter(isLeap, yrs)
```

```
[1792, 2000, 2012]
```

The `filter` function applies `isLeap` to each element of `yrs` and returns only those for which the result is `true`.

Since the solution was pretty easy let's add some more tests to let our fingers cool down slowly. If you read the Wikipedia's page[52]carefully then you know that for every 400 years period we got 303 regular years and 97 leap years. Let's see if that rule holds.

```
function runTestSet()::Int
    startYr::Int = 0
    endYr::Int = 0
    numLeapYrs::Int = 0
    for i in 1:3601
        startYr = i
        endYr = i + 400 - 1
        numLeapYrs = filter(isLeap, startYr:endYr) |> length
        if numLeapYrs != 97
            return 1
        end
    end
    return 0 # C-like return value
end

runTestSet()
```

0

Here we tested different time periods (each spanning 400 years). Notice that last such a period starts in the year `3601`, since our function handles inputs in the range [1 - 4000] and `length(3600:4000)` is actually equal to 401. Anyway, for each of the periods, we counted the number of leap years (`numLeapYrs`) with `filter` and `length`. If `numLearpYrs` differs from `97`, we return `1` right away (and skip other checks). Otherwise (once all the periods passed the tests) we return `0`. The above is a convention met in C

[52]https://en.wikipedia.org/wiki/Gregorian_calendar

programming language (in the `main` function). It is sufficient for our simplistic case, however, for more serious applications you should follow the testing advice contained in the docs[53].

[53]https://docs.julialang.org/en/v1/stdlib/Test/

Calendar

In this chapter I used the following libraries.

```
import Dates as Dt # internal library
```

Still, once you read the problem description you may decide to do otherwise.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[54](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[55].

Problem

A Gnu/Linux operating system comes with a bunch of utilities[56]that help with an everyday life. One of them is cal[57]command that displays a calendar. Let's try to mimic a fraction of its power. Write a program that prints a calendar for a given month that looks similarly to:

```
> cal Jun 2025 # shell command (output starts 1 line below)
     June 2025
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30
```

Use it to tell on which day of the week you were born.

Alternatively, assume that the Gregorian calendar has been applied throughout the whole Common Era[58]and based on that say:

---

[54]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/calendar
[55]https://docs.julialang.org/en/v1/stdlib/Pkg/
[56]https://en.wikipedia.org/wiki/Util-linux
[57]https://en.wikipedia.org/wiki/Cal_(command)
[58]https://en.wikipedia.org/wiki/Common_Era

- on what day of the week was Jesus born (assume: Dec 25, year 1)?
- on what day of the week was the world supposed to come to an end (assume: Dec 21, year 2012, but you got plenty dates to choose from[59])?
- on what day of the week will the next millennium start (assume: Jan 1, 3000)?

Try not to employ the built-in Dates[60]module in your solution (unless you have to). Still, you may use it to verify your results, e.g. in order to know on which day did this year start just type:

```
import Dates as Dt

d = Dt.Date(2025, 1, 1)
Dt.dayname(d)
```

Wednesday

BTW. You may use the above day as a reference point.

Solution

First, let's begin by defining a few rather self explanatory constants (`const`) that we will use throughout our program.

```
const DAYS_PER_WEEK = 7
const DAYS_PER_MONTH = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
const DAYS_PER_MONTH_LEAP = [31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30,
31]
const SHIFT_YR = 365
const SHIFT_YR_LEAP = 366
const WEEKDAYS_NAMES = ["Su", "Mo", "Tu", "We", "Th", "Fr", "Sa"]
const MONTHS_NUM_2_NAME = Dict(
    1 => "January", 2 => "February", 3 => "March",
    4 => "April", 5 => "May", 6 => "June", 7 => "July",
    8 => "August", 9 => "September", 10 => "October",
    11 => "November", 12 => "December")
const MONTHS_NAME_2_NUM = Dict(
    "Jan" => 1, "Feb" => 2, "Mar" => 3,
    "Apr" => 4, "May" => 5, "Jun" => 6, "Jul" => 7,
```

[59]https://en.wikipedia.org/wiki/List_of_dates_predicted_for_apocalyptic_events
[60]https://docs.julialang.org/en/v1/stdlib/Dates/

```
    "Aug" => 8, "Sep" => 9, "Oct" => 10,
    "Nov" => 11, "Dec" => 12)
```

> Note. Using `const` with mutable containers like vectors or dictionaries allows to change their contents later on, e.g., with `push!`. So the `const` used here is more like a convention, a signal that we do not plan to change the containers in the future. If we really wanted an immutable container then we should consider a(n) (immutable) tuple. Anyway, some programming languages suggest that `const` names should be declared using all uppercase characters to make them stand out. Here, I follow this convention.

As you can see from the output of `cal Jan 2025` (see Section 8.1) we get a rectangular printout with 7 columns and `x` rows. Clearly, the number of elements is a multiple of 7. So, let's write a function that determines how many elements should be in our rectangle.

```
function getMultOfYGtEqX(x::Int, y::Int=DAYS_PER_WEEK)::Int
    @assert x > 0 && y > 0 "x and y must be > 0"
    @assert x >= y "x must be >= y"
    q::Int, r::Int = divrem(x, y) # quotient, reminder
    return r == 0 ? x : y*(q+1)
end
```

To that end we wrote `getMultOfYGtEqX` that, as its name implies, returns the multiple of `y` that is greater than or equal to `x`. First, thanks to `divrem` function, we get the quotient (`q` - the number of 'full' `y`s is inside of `x`) and the reminder (`r` - the rest after the integer division) after dividing `x` by `y`. If `x` is evenly divisible by `y` (`r == 0 ?`) then our result is just `x` (`x` is the multiple of `y`). Otherwise, we multiply `y` by the quotient plus 1 (`y*(q+1)`).

We will use it (`getMultOfYGtEqX`) to get our days for a given month padded with zeros.

```
# 1 - Sunday, 7 - Saturday
function getPaddedDays(nDays::Int, fstDay::Int)::Vec{Int}
```

```
    @assert 0 < fstDay < 8 "fstDay must be in range [1-7]"
    daysFront::Int = fstDay - 1
    days::Vec{Int} = zeros(getMultOfYGtEqX(nDays+daysFront,
DAYS_PER_WEEK))
    days[fstDay:(fstDay+nDays-1)] = 1:nDays
    return days
end

function vec2matrix(v::Vec{T}, r::Int, c::Int,
                    byRow::Bool)::Matrix{T} where T
    @assert (r > 0 && c > 0) "r and c must be positive integers"
    @assert (length(v) == r*c) "length(v) must be equal to r*c"
    m::Matrix{T} = Matrix{T}(undef, r, c)
    stepBegin::Int = 1
    stepSize::Int = (byRow ? c : r) - 1
    for i in 1:(byRow ? r : c)
        if byRow
            m[i, :] = v[stepBegin:(stepBegin+stepSize)]
        else
            m[:, i] = v[stepBegin:(stepBegin+stepSize)]
        end
        stepBegin += (stepSize + 1)
    end
    return m
end
```

All that we need for that is to know the number of days in a given month (`nDays`) and what is the first day (`fstDay`, where 1 is Sunday and 7 is Saturday). We use the above as the arguments to `getPaddedDays`. The function creates a vector of `zeros` that contains number of elements that is a multiple of 7 (`DAYS_PER_WEEK`). The vector length is determined by `getMultOfYGtEqX` and is at least `nDays+daysFront` long. We fill the vector (starting at `fstDay`) with digits for all the days (`1:nDays`). Finally, we return `days`.

Afterwards, we want to put the vector (result of `getPaddedDays`) into a matrix with 7 columns (`DAYS_PER_WEEK`) and the appropriate number of rows. For that we wrote `vec2matrix`, that unlike the built in reshape[61], will allow us to put the vector (`v`) into the matrix (`m`) row by row (when `byRow = true`).

Let's see how it works for January 2025.

[61]https://docs.julialang.org/en/v1/base/arrays/#Base.reshape

```
jan2025 = getPaddedDays(31, 4)
vec2matrix(jan2025, Int(length(jan2025) / DAYS_PER_WEEK),
    DAYS_PER_WEEK, true)
```

```
5×7 Matrix{Int64}:
  0   0   0   1   2   3   4
  5   6   7   8   9  10  11
 12  13  14  15  16  17  18
 19  20  21  22  23  24  25
 26  27  28  29  30  31   0
```

Pretty good, and how about this month (June 2025).

```
jun2025 = getPaddedDays(30, 1)
vec2matrix(jun2025, Int(length(jun2025) / DAYS_PER_WEEK),
    DAYS_PER_WEEK, true)
```

```
5×7 Matrix{Int64}:
  1   2   3   4   5   6   7
  8   9  10  11  12  13  14
 15  16  17  18  19  20  21
 22  23  24  25  26  27  28
 29  30   0   0   0   0   0
```

It appears to be working as intended.

Time to format that output a little.

```
# 1 - Sunday, 7 - Saturday
function getFmtMonth(fstDayMonth::Int, nDaysMonth::Int,
                     month::Int, year::Int)::Str
    @assert 1 <= fstDayMonth <= 7 "fstDayMonth must be in range [1-7]"
    @assert 28 <= nDaysMonth <= 31 "nDaysMonth must be in range [28-31]"
    @assert 1 <= month <= 12 "month must be in range [1-12]"
    @assert 1 <= year <= 4000 "year must be in range [1-4000]"
    topRow2::Str = join(WEEKDAYS_NAMES, " ")
    topRow1::Str = center(
        string(MONTHS_NUM_2_NAME[month], " ", year), length(topRow2))
    days::Vec{Str} = string.(getPaddedDays(nDaysMonth, fstDayMonth))
    days = replace(days, "0" => " ")
    m::Matrix{Str} = vec2matrix(
        days, Int(length(days)/daysPerWeek), DAYS_PER_WEEK, true)
    fmtDay(day) = lpad(day, 2)
    fmtRow(row) = join(map(fmtDay, row), " ")
    result::Str = ""
```

```
    for r in eachrow(m)
        result *= fmtRow(r) * "\n"
    end
    return topRow1 * "\n" * topRow2 * "\n" * result
end
```

We begin with a couple of sanity checks (`@assert` statements). Next, we `join` `WEEKDAYS_NAMES` into a one long string separated with spaces (`" "`) to get a row just on top of the days (`topRow2`). Above that (`topRow1`) we will place a month name (`MONTHS_NUM_2_NAME[month]`) and a `year` (like `"January 2025"`), which we `center` with the function developed in Section 18.2 . Finally, the consecutive rows will be occupied by `days` written with the Arabic numerals and expressed as vector of strings (`days::Vec{Str}`). However, we replace the zeros (`"0"` used for padding) with spaces and put them (`days`) into a matrix (`m`). All that's left to do is to define day (`fmtDay`) and row (`fmtRow`) formatters (inline functions) for our matrix. We proceed by building our `result` row by row (`result *= fmtRow(r) * "\n"`). Finally, we return a formatted month by gluing everything together (`topRow1 * "\n" * topRow2 * "\n" * result`). Let's take a sneak peak.

January 2025:

```
getFmtMonth(4, 31, 1, 2025)
    January 2025
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

and June 2025:

```
getFmtMonth(1, 30, 6, 2025)
      June 2025
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
```

```
22 23 24 25 26 27 28
29 30
```

At this point we're basically done. That is, if we wanted to take a shortcut and rely on the built-in `Dates` module to calculate a day of the week for us (for details see Section 8.1). Of course, a(n) (over)zealous Julia programmer would never do no such a thing. So let's try to figure it out on our own with `getShiftedDay`.

```
# 1 - Sunday, 7 - Saturday
function getShiftedDay(curDay::Int, by::Int)::Int
    @assert 1 <= curDay <= 7 "curDay not in range [1-7]"
    newDay::Int = curDay
    shift::Int = abs(by) % DAYS_PER_WEEK
    move::Int = by < 0 ? -1 : 1
    for _ in 1:shift
        newDay += move
        newDay = newDay < 1 ? 7 :
            (newDay > 7 ? 1 : newDay)
    end
    return newDay
end
```

The function accepts the current day (`curDay`) and a shift (`by`). That last parameter is the number of days before (negative values) or after (positive values) `curDay`. The actual `shift` is calculated using the modulo operator[62](`%`), since a shift by let's say +15 days is actually a shift by two weeks (which we may ignore) and 1 day (`abs(15) % 7` is 1). Next, we make as many moves (day before is `-1`, day after is `1`) as indicated by `shift` (`for _ in 1:shift`). However, if we stepped out of the range to the left (`newDay < 1 ?`), we begin from the other side (`7`th day of the week). Alternatively (`:`), if we stepped out of range to the right (`newDay > 7 ?`), we begin from the start (`1`). Otherwise, we leave `newDay` as it was (`: newDay`). Notice, however, that if `shift` is equal to 0 then the code in the `for` loop will not be executed and `newDay` equal to `curDay` will be returned (which is what we want, e.g. for `by = 0` or `by = 14`).

[62]https://docs.julialang.org/en/v1/base/math/#Base.rem

Now, `getFmtMonth` and `getPaddedDays` require a day of the week with which a month starts plus the number of days in that month. Let's use our `getShiftedDay` to calculate that for any month in a given year.

```
# 1 - Sunday, 7 - Saturday
# returns (1st day of month, num of days in this month)
function getMonthData(dayJan1::Int, month::Int, leap::Bool)::Tuple{Int,
Int}
    @assert 1 <= dayJan1 <= 7 "day not in range [1-7]"
    @assert 1 <= month <= 12 "month not in range [1-12]"
    curDay::Int = dayJan1
    daysInMonths::Vec{Int} = leap ? DAYS_PER_MONTH_LEAP : DAYS_PER_MONTH
    if month == 1
        return (dayJan1, daysInMonths[month])
    end
    for m in 2:month
        curDay = getShiftedDay(curDay, daysInMonths[m-1])
    end
    return (curDay, daysInMonths[month])
end
```

The function is rather simple. If we want to know when a given month begins we just shift `curDay` (initialized with `dayJan1`) by as many days as there were in the previous month (`daysInMonths[m-1]`) for all the previous months up to this one (`for m in 2:month`).

If we can do such a shift for a month in a given year, we can also do it for any month in any year.

```
# 1 - Sunday, 7 - Saturday
# returns (1st day of month, num of days in this month)
function getMonthData(yr::Int, month::Int)::Tuple{Int, Int}
    @assert 1 <= yr <= 4000 "yr not in range [1-4000]"
    @assert 1 <= month <= 12 "month not in range [1-12]"
    curDay::Int = 4 # 1st Jan 2025 (reference point) was Wednesday
    start::Int = yr <= 2025 ? 2025-1 : 2025+1
    step::Int = yr <= 2025 ? -1 : 1
    yrShift::Int = 0
    for y in start:step:yr
        yrShift = isLeap(y) ? SHIFT_YR_LEAP : SHIFT_YR
        curDay = getShiftedDay(curDay,  yrShift * step)
    end
    return getMonthData(curDay, month, isLeap(yr))
end
```

We begin, by setting a reference point (`curDay`) to be January 1, 2025 (let's say that we've got a calendar on a wall in front of us that we can rely on for that information). Next, thanks to the for loop (and `getShiftedDay`) we figure out on which day of the week a given year (`yr`) starts (we get there year by year with `for y in start:step:yr`). Of course, we take into account leap years (see `isLeap` from Section 7.2) if there are any. Finally (`return`), we use the previously defined `getMonthData` method, to get the necessary information (starting day, days in month) for a month we are looking for.

All that's left to do is to pack it all into `getCal` wrapper for the ease of use.

```
function getCal(month::Int, yr::Int)::Str
    # ... - unpacks tuple into separate values
    return getFmtMonth(getMonthData(yr, month)..., month, yr)
end

function getCal(month::Str, yr::Int)::Str
    m::Int = monthsName2Num[month]
    return getCal(m, yr)
end
```

Time for the first swing.

```
getCal("Jan", 2025)
    January 2025
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

And now for the questions.

On what day of the week was Jesus born (assume: Dec 25, year 1)?

```
getCal("Dec", 1)
     December 1
Su Mo Tu We Th Fr Sa
                   1
 2  3  4  5  6  7  8
```

```
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

On what day of the week was the world supposed to come to an end (assume: Dec 21, year 2012)?

```
getCal("Dec", 2012)
    December 2012
Su Mo Tu We Th Fr Sa
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

On what day of the week will the next millennium start (assume: Jan 1, 3000)?

```
getCal("Jan", 3000)
    January 3000
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

Of course, the above results should be correct under the assumption that the Gregorian calendar[63]has been uniformly applied throughout the whole Common Era[64]. Needless to say, that was not the case, so I wouldn't rely on `getCal` too much for your time travel.

[63]https://en.wikipedia.org/wiki/Gregorian_calendar
[64]https://en.wikipedia.org/wiki/Common_Era

Prime Numbers

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[65](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[66].

Problem

A prime number is an integer greater than 1 that is divisible only by 1 and itself. Prime numbers found practical applications, e.g. in cryptography.

In this task your job is to write a function that returns all prime numbers up to a certain extent. You may use any computational method from the Wikipedia's page[67].

Solution

Trial Division

We begin by implementing trial division[68]algorithm. The above is a 'brute force' approach, namely we divide our candidate number, `n`, by a series of integers (remember, a prime number is an integer greater than 1 that is divisible only by 1 and itself). The method has a small improvement, i.e. the divisors are in the range from 2 up to $\sqrt{n}$.

```
function isPrime(n::Int)::Bool
    if n < 4
```

[65]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/prime_numbers

[66]https://docs.julialang.org/en/v1/stdlib/Pkg/

[67]https://en.wikipedia.org/wiki/Prime_number#Computational_methods

[68]https://en.wikipedia.org/wiki/Prime_number#Trial_division

```
        return n < 2 ? false : true
    end
    upTo::Int = sqrt(n) |> ceil |> Int
    for i in 2:upTo
        if n % i == 0
            return false
        end
    end
    return true
end
```

The function is pretty straightforward. The first two prime numbers are 2 and 3. Hence, the if `n < 4` block that uses a ternary expression[69]. Next, per algorithm we establish the maximum value of the divisor (`upTo`). To that end we calculate `sqrt(n)`, round it to up to the next whole number (`ceil`) and convert it to integer (`Int`). Then, we test if the divisors in the range `2:upTo` divide our number `n` evenly (`n % i == 0`), if so we `return false` right away, otherwise the number is prime (`return true`).

With our `isPrime` ready, generating a sequence of primes is a breeze.

```
function getPrimesV1(upTo::Int)::Vec{Int}
    @assert upTo > 1 "upTo must be > 1"
    return filter(isPrime, 2:upTo)
end
```

All we had to do was to `filter` out primes from the sequence of numbers within the desired range `2:upTo`. Time for a simple test.

```
# prime numbers up to 100 from:
# https://en.wikipedia.org/wiki/List_of_prime_numbers
primesFromWiki = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]

getPrimesV1(100) == primesFromWiki
```

true

Works like a charm.

[69]https://b-lukaszuk.github.io/RJ_BS_eng/julia_language_decision_making.html#sec:ternary_expression

Sieve of Eratosthenes

Next, we will implement the sieve of Eratosthenes algorithm[70]that finds primes by

> … iteratively marking as composite (i.e., not prime) the multiples of each prime, starting with the first prime number, 2.

(see the Wikipedia's page from the link above).

Let's get down to it.

```
function getPrimesV2(upTo::Int)::Vec{Int}
    @assert upTo > 1 "upTo must be > 1"
    nums::Vec{Int} = 1:upTo |> collect
    isPrimeTests::Vec{Bool} = ones(Bool, upTo)
    isPrimeTests[1] = false # first prime is: 2
    for num in nums
        if isPrimeTests[num]
            # numMultiples - local variable visible only in for loop
            numMultiples = (num*2):num:upTo # multiples of num
            isPrimeTests[numMultiples] .= false
        end
    end
    return nums[isPrimeTests]
end
```

This time we begin by defining two vectors `nums` which contains all the examined integers from a given range and `isPrimeTests` that contains indication of whether a number in `nums` is prime. Initially, all the numbers are considered to be primes (`ones(Bool, upTo)`) except for 1 (`isPrimeTests[1] = false`). Then, we examine every number (`num`) out of the candidates (`nums`). If a number is prime (`if isPrimeTests[num]`) we set all its multiples (`numMultiples`) as not primes (`false`) using broadcasting (`.`) with the assignment (`=`) operator. Once finished we return only the candidates that passed the test (`return nums[isPrimeTests]`) using boolean indexing of a vector.

Time for a simple test.

[70]https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

```
getPrimesV2(100) == primesFromWiki
```

true

The test passed. Before we close this chapter let's see if the two functions give identical results.

```
getPrimesV1(1000) == getPrimesV2(1000)
```

true

Yes, they do.

Recursion

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[71](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[72].

Problem

Recursion[73]is a programming technique where a function invokes itself in order to solve a problem. It relies on two simple principles:

1. know when to stop
2. split the problem into a single step and a smaller problem

And that's it. Let's see how this works in practice.

For that we'll write a recursive function that sums the elements of a vector of integers.

```
function recSum(v::Vec{Int})::Int
    if isempty(v)
        return 0
    else
        return v[1] + recSum(v[2:end])
    end
end
```

We begin by defining our edge case (know when to stop). If the vector (`v`) is empty (`if isempty(v)`) we return 0 (BTW. Notice that zero is a neutral mathematical operation for addition, any number plus zero is just itself). Otherwise (`else`) we add the first element of the vector

[71]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/recursion
[72]https://docs.julialang.org/en/v1/stdlib/Pkg/
[73]https://en.wikipedia.org/wiki/Recursion_(computer_science)

(`v[1]`, single step) to whatever number is returned by `recSum` with the rest of the vector (`v[2:end]`) as an argument (smaller problem).

Time for a couple of examples.

For

```
recSum([1])
```

1

the execution of the program looks something like

```
recSum([1]) # triggers else branch
    1 + recSum([]) # triggers if branch (stop)
        1 + 0 # stop reached, time to return
1
```

Whereas for

```
recSum([1, 2, 3])
```

6

we get

```
recSum([1, 2, 3]) # triggers else branch
    1 + recSum([2, 3]) # triggers else branch
        1 + 2 + recSum([3]) # triggers else branch
            1 + 2 + 3 + recSum([]) # triggers if branch (stop)
                1 + 2 + 3 + 0 # stop reached, time to return
6
```

It is difficult to imagine step by step, but `recSum` works equally well for a bit broader range of numbers.

```
1:100 |> collect |> recSum
```

5050

Believe it or not, but that last one you can calculate fairly easily

yourself if you apply the Gauss method[74].

Anyway, usually recursion is not very effective and it can be rewritten with loops. Moreover, too big input, e.g. `1:100_000`, will likely cause an error with `recSum`, but not with the built-in `sum`. Still, for some problems recursion is an easy to implement and elegant solution that gets the job done. Therefore, it is worth to have this technique in your programming toolbox.

Classical examples of recursive process in action are the factorial[75]and the Fibonacci sequence[76]which are your tasks for this section.

Solution

A factorial is an interesting little function with a set of practical applications, one of them I explained here[77].

Its recursive implementation follows closely the mathematical definition (see below, where `n!` is a factorial of a number `n`).

$$n! = \{1 \quad \text{if} \quad n = 1 \\ n \times (n-1)! \quad \text{otherwise}$$

Which can be translated into Julia's:

```
function recFactorial(n::Int)::Int
    @assert 1 <= n <= 20 "n must be in range [1-20]"
    if n == 1
        return 1
    else
        return n * recFactorial(n-1)
    end
end
```

In general, a factorial is well defined for positive integers and it grows

[74]https://www.nctm.org/Publications/TCM-blog/Blog/The-Story-of-Gauss/
[75]https://en.wikipedia.org/wiki/Factorial
[76]https://en.wikipedia.org/wiki/Fibonacci_sequence
[77]https://b-lukaszuk.github.io/RJ_BS_eng/statistics_intro_exercises.html#sec:statistics_intro_exercise2

very quickly (n > 20 would produce overflow[78], to resolve it we could use `BigInt` instead of `Int`) hence the `@assert` line. Anyway, for `n` equal 1 we return 1 (our base case), otherwise we multiply `n` by `recFactorial(n-1)`. Go ahead, follow the execution of the program for small inputs like `recFactorial(3)` in your head (similarly to `recSum` from Section 10.1).

To get you a better feel for recursion in Julia, here are two other equivalent implementations of `recFactorial`.

```
function recFactorialV2(n::Int)::Int
    @assert 1 <= n <= 20 "n must be in range [1-20]"
    return n == 1 ? 1 : n * recFactorialV2(n-1)
end
```

and

```
function recFactorialV3(n::Int, acc::Int=1)::Int
    @assert 1 <= n <= 20 "n must be in range [1-20]"
    return n == 1 ? acc : recFactorialV3(n-1, n * acc)
end
```

The second version (`recFactorialV2`) uses ternary operator[79]instead of more verbose `if else` statements. The third version (`recFactorialV3`) relies on a so called accumulator (`acc`) that stores the result of a previous calculation (if any). `recFactorialV3` is a tail-recursive function that is recommended in some programming languages, like Haskell[80]and Scala[81], that can take advantage of this kind of code to produce (internally) an effective function implementation.

Let's go to the Fibonacci sequence. When I was a student, they said that dinners in the student's canteen are like Figonacci numbers,

[78]https://docs.julialang.org/en/v1/manual/integers-and-floating-point-numbers/#Overflow-behavior

[79]https://docs.julialang.org/en/v1/base/base/#?:

[80]https://en.wikipedia.org/wiki/Haskell

[81]https://en.wikipedia.org/wiki/Scala_(programming_language)

i.e. each dinner is the sum of the two previous ones. To put it more mathematically, we get

$$fib(n) = \{0 \quad \text{if } n = 0 \\ 1 \quad \text{if } n = 1 \\ fib(n-2) + fib(n-1) \quad \text{otherwise}$$

Which expressed in Julia gives us:

```
function recFib(n::Int)::Int
    @assert 0 <= n <= 40 "n must be in range [0-40]"
    if n == 0
        return 0
    elseif n == 1
        return 1
    else
        return recFib(n - 2) + recFib(n - 1)
    end
end

recFib(10)
```

55

The numbers do not grow as fast as `factorials`, but the algorithm, although simple, is very inefficient. For instance, for `recFib(3)` I have to calculate `recFib(1) + recFib(2)`, but `recFib(2)` will calculate `recFib(1)` inside of it as well. For greater numbers (inputs) the duplicated operations threaten to throttle the processor. On my laptop the computation for `recFib(40)` takes roughly 600-700 [ms], so more than half a second, a delay noticed even by a human.

Therefore we may improve our last function by using lookup tables/ dictionaries like so:

```
function recFib!(n::Int, lookup::Dict{Int, Int})::Int
    @assert 0 <= n <= 40 "n must be in range [0-40]"
    @assert(haskey(lookup, 0) && haskey(lookup, 1),
        "lookup must have base cases")
    if !haskey(lookup, n)
        lookup[n] = recFib!(n-2, lookup) + recFib!(n-1, lookup)
    end
```

```
    return lookup[n]
end
```

Notice that the function modifies `lookup` (hence `!` appended to its name, per Julia's convention). Inside we check if there is a value for a Fibonacci's number in `lookup`. If not (`!haskey(lookup, n)`) then we calculate it using the known formula and insert it in `lookup`. Otherwise we just return the found value.

The last function (`recFib!`) is more performant, as:

```
fibs = Dict(0 => 0, 1 => 1)
recFib!(40, fibs)
```

102334155

takes only microseconds on its first execution (hundreds to thousand times faster). Interestingly, running `recFib!(40, fibs)` for the second time reduces the time to nanoseconds (million(s) times faster) since there are no calculations performed the second time, just reading the number from the previously modified `lookup`). Run `recFib(40)` twice to convince yourself that it takes roughly the same amount of time every time it runs with the same `n`.

Randomness

In this chapter I used the following libraries.

```
import Dates as Dt # internal library
import Statistics as St # internal library
```

Still, once you read the problem description you may decide to do otherwise.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[82](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[83].

Problem

Random numbers are the numbers that occur, well, at random. Anyway, they are quite useful for programming. Chances are you will use them to solve some problems like those in Section 12.1 or Section 13.1 .

But how do computers generate them. Surprise, they don't. But then how could a Julia's `Random.jl` (part of the standard library) generate one. Hmm, it's complicated, but the basic idea could be explained with an example.

In a moment I will use a function that generates a random number in the range of 1 to 10 (inclusive-inclusive). Take a moment and try to guess it.

Ready, here we go.

```
getRand()
```

[82]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/randomness

[83]https://docs.julialang.org/en/v1/stdlib/Pkg/

The number was seven. Did you guess it? If so, congrats! The popular psychology claims that for the said range people most often choose 7 and 3 so I wonder just how many of you, the readers, succeeded. Anyway, let me try again, I will use comprehensions[84]to generate 10 random numbers and you will try to guess the next one, ready.

```
[getRand() for _ in 1:10]
```

```
[8, 9, 10, 1, 2, 3, 4, 5, 6, 7]
```

OK, what's the next number?

You got it, it's 8. Now let's examine this sub-optimal number generator.

```
seed = 6

function getRand()::Int
    newSeed::Int = (seed % 10) + 1
    global seed = newSeed
    return newSeed
end
```

First, we declare a variable called `seed` with an initial, undisclosed and hard to predict value. From this `seed` our random numbers will sprout. Next, inside `getRand` we update the `seed` using some mathematical formula and return it as our result. Moreover, we update the old `seed` with that new value (`global seed = newSeed`, we use global[85]since here `seed` is in the global scope[86]). Thanks to the update the next time we use `getRand` it will return us some new value. Still, the function is purely deterministic as once you know the value of `seed` you can accurately predict the random variable you will get.

[84]https://b-lukaszuk.github.io/RJ_BS_eng/julia_language_repetition.html#sec:julia_language_comprehensions

[85]https://docs.julialang.org/en/v1/base/base/#global

[86]https://docs.julialang.org/en/v1/manual/variables-and-scoping/#scope-of-variables

Surprisingly this is more or less what the random number generators[87] do. The idea we got there was not bad, we just need to improve on its execution. We need a less obvious starting value, a more chaotic update method, and a much longer looping period so that no one sees that the sequence repeats itself periodically.

And here we are. In this chapter we are going to develop a simplified library for random numbers generation.

First, pick a random number generator[88]that is relatively easy to implement (LCG[89]seems to be good enough, but it's up to you). The `seed` is often initialized with epoch time[90], e.g. the number of seconds that passed since 1 January 1970, which is kind of unpredictable. You should be able to get it out of the Dates[91]module. Use the LCG to write `getRand` that returns a random `Float64` in the range `[0-1)` (inclusive - exclusive). This is our bread and butter of random number generation. It works similar to JavaScript's `Math.random()` or Julia's `rand()` (`Base.rand` imports it from `Random.jl`). Next, define another `getRand` method, the one that returns a random integer from a given range. To cool down read about Box-Mueller transform[92]and use it to define two `getRandn` methods, one that returns a value from a standard normal distribution[93]with the mean = 0 and the standard deviation = 1 (like `Base.randn` do). The other should return a normal distribution with a specified mean and standard deviation.

If the above feels overwhelming, then proceed one step at a time and do as much as you can. Remember you need to understand this stuff enough to implement it in the code, leave the theorems and proofs to mathematicians and trust that they did their jobs right.

---

[87]https://en.wikipedia.org/wiki/Random_number_generation

[88]https://en.wikipedia.org/wiki/List_of_random_number_generators#Pseudorandom_number_generators_(PRNGs)

[89]https://en.wikipedia.org/wiki/Linear_congruential_generator

[90]https://en.wikipedia.org/wiki/Epoch_(computing)

[91]https://docs.julialang.org/en/v1/stdlib/Dates/

[92]https://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform

[93]https://en.wikipedia.org/wiki/Normal_distribution#Standard_normal_distribution

Solution

We start by defining a few global variables required by LCG as well as a a way to set our `seed` to a desired value.

```
import Dates as Dt

a = 1664525
c = 1013904223
m = 2^32
seed = round(Int, Dt.now() |> Dt.datetime2unix)

function setSeed!(newSeed)::Nothing
    global seed = newSeed
    return nothing
end
```

The values were chosen based on this table from Wikipedia[94].

Time to translate the LCG algorithm to Julia's code.

```
function getRandFromLCG()::Int
    newSeed::Int = (a * seed + c) % m
    setSeed!(newSeed)
    return newSeed
end
```

> ***Note:*** In general a function should not rely on nor change global variables. Instead it should only depend on the parameters that were send to it as its arguments. Relying on global variables although convenient and tempting could lead to bugs that are hard to pinpoint and eliminate. That's why in the rest of this book I will try to avoid this style with the exception of global constant variables seen, e.g. in Section 8.2 .

Let's see how it works.

```
# your numbers will likely differ from mine
[getRandFromLCG() for _ in 1:6]
```

[94]https://en.wikipedia.org/wiki/Linear_congruential_generator#Parameters_in_common_use

```
[955681523, 3993752502, 3056154525, 1459901528, 2648329175, 2730194762]
```

A small victory. In result we got some integers that are very hard to predict for a human brain alone.

Still, the numbers are unwieldy and look quite odd. How can we transform them into a function that returns `Float64` from the range `[0-1)`? The key is the modulo operator (`%` equivalent to `rem` function) in `getRandFromLCG`. All it is is just a reminder after a division. It has an interesting property, the remainder of `i` divided by `m` is always in the range 0 to `m-1`, see the example below.

```
[i % 3 for i in 1:7]
```

```
[1, 2, 0, 1, 2, 0, 1]
```

If so then all we have to do is to divide our `seed` (that takes values from 0 to `m`−1) by `m` to get the desired `Float64` value.

```
function getRand()::Flt
    return getRandFromLCG() / m
end

# your numbers will likely differ from mine
[getRand() for _ in 1:3]
```

```
[0.8426131086889654, 0.8208084730431437, 0.4596601116936654]
```

Much better, we reached the first checkpoint. OK, now how to convert it to a random integer generator. Let's try one step at a time. The `getRand()` returns a `Float64` in the range `[0-1)`. So, if we were to multiply 1 (the upper limit) by let's say 5 we would get 5, and 0 (the lower limit) times 5 would get us zero. Hence, the following code (`floor` returns the nearest integer smaller than or equal to a `Float64`).

```
function getRand(upToExcl::Int)::Int
    @assert 0 < upToExcl "upToExcl must be greater than 0"
```

```
    return floor(getRand() * upToExcl)
end
```

Time for a test. If we did our job right we should get a sequence of random values each equally likely to occur (`getCounts` was developed and explained here[95]).

```
function getCounts(v::Vec{T})::Dict{T,Int} where T
    counts::Dict{T,Int} = Dict()
    for elt in v
        counts[elt] = get(counts, elt, 0) + 1
    end
    return counts
end

setSeed!(1111) # for reproducibility
[getRand(3) for _ in 1:100_000] |> getCounts
```

```
Dict{Int64, Int64} with 3 entries:
  0 => 33421
  2 => 33054
  1 => 33525
```

Yep, roughly equal counts. One more swing with a different range.

```
setSeed!(1111) # for reproducibility
[getRand(5) for _ in 1:100_000] |> getCounts
```

```
Dict{Int64, Int64} with 5 entries:
  0 => 20046
  4 => 19821
  2 => 20111
  3 => 19882
  1 => 20140
```

Ladies and gentlemen, we got it. Now, let's tweak it a bit so that we can get an integer in the desired range.

[95] https://b-lukaszuk.github.io/RJ_BS_eng/statistics_prob_theor_practice.html

```
function getRand(minIncl::Int, maxIncl::Int)::Int
    @assert 0 < minIncl < maxIncl "must get: 0 < minIncl < maxIncl"
    return minIncl + getRand(maxIncl-minIncl+1)
end

function getRand(n::Int, minIncl::Int, maxIncl::Int)::Vec{Int}
    @assert 0 < n "n must be greater than 0"
    return [getRand(minIncl, maxIncl) for _ in 1:n]
end
```

For that we just generated an `Int` from 0 up to one more than the span of our range (`maxIncl-minIncl+1`) and added `minIncl` so that the range starts at that value and not at zero. Let's check it out.

```
setSeed!(2222)
getRand(100_000, 1, 4) |> getCounts
```

```
Dict{Int64, Int64} with 4 entries:
  4 => 24969
  2 => 24635
  3 => 25162
  1 => 25234
```

Looks good, roughly equal counts was what we were looking for. One more time, just to be sure.

```
setSeed!(2222)
getRand(100_000, 3, 7) |> getCounts
```

```
Dict{Int64, Int64} with 5 entries:
  5 => 19944
  4 => 19722
  6 => 20224
  7 => 19874
  3 => 20236
```

Another checkpoint reached.

As for the Box-Mueller transform there is a small problem. The Wikipedia's page already contains the algorithm implementation in

Julia[96]. So for a change we will take the JavaScript version[97] and translate it to Julia.

```
function getRandn()::Tuple{Flt, Flt}
    theta::Flt = 2 * pi * getRand()
    R::Flt = sqrt(-2 * log(getRand()))
    x::Flt = R * cos(theta)
    y::Flt = R * sin(theta)
    return (x, y)
end
```

And voila, the only difference is that instead of a vector we return a tuple with each element being a value from the normal distribution with the mean = 0 and the standard deviation = 1. However, we would prefer a function that returns any number of normally distributed values (and not only two in `Tuple{Flt, Flt}`), hence the following code.

```
function flatten(randnNums::Vec{Tuple{Flt, Flt}})::Vec{Flt}
    len::Int = length(randnNums) * 2
    result::Vec{Flt} = Vec{Flt}(undef, len)
    i::Int = 1
    for (a, b) in randnNums
        result[i] = a
        result[i+1] = b
        i += 2
    end
    return result
end

function getRandn(n::Int)::Vec{Flt}
    @assert n > 0 "n must be greater than 0"
    roughlyHalf::Int = cld(n, 2)
    return flatten([getRandn() for _ in 1:roughlyHalf])[1:n]
end
```

`getRandn(n::Int)` generates a vector of tuples using comprehensions. The length of the vector is determined using `cld`, which divides `n` by 2, and returns the smallest integer equal to or greater than the result of that division. The vector of tuples (in this case `Vec{Tuple{Flt, Flt}}`) is `flatten`ed before being returned from `getRandn(n::Int)`. That's

[96] https://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform#Julia
[97] https://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform#JavaScript

why we get a regular vector of floats (`Vec{Flt}`). Notice, that if `n` is an odd number then we return all but the last element (`[1:n]`) of the vector (since `flatten` returns a vector of even length). Anyway, let's see how we did.

```
import Statistics as St

# test: mean ≈ 0.0, std ≈ 1.0
setSeed!(401)
x = getRandn(100)

St.mean(x), St.std(x)
```

```
(0.050968498407633414, 1.0266071355197548)
```

I would say we did a pretty good job.

Time for the last step, let's transform `getRandn` to a function that provides a normal distribution with a specified mean and standard deviation. That's fairly simple. Since the 'original' deviation is 1, then if we multiply the numbers by let's say 16, then we will get a distribution with the mean 0 and standard deviation equal to 16. So how do we make a mean equal, let's say 100? It's easy as well, we just add 100 to every number from a distribution.

```
function getRandn(n::Int, mean::Flt, std::Flt)::Vec{Flt}
    return mean .+ std .* getRandn(n)
end

# test: mean ≈ 100.0, std ≈ 16.0
setSeed!(401)
x = getRandn(100, 100.0, 16.0)

St.mean(x), St.std(x)
```

```
(100.81549597452215, 16.425714168316077)
```

That seems to be it, we reached the end of this task. Together we developed a simplified library for random numbers generation and it wasn't all that bad, was it? Still, for practical reasons I would

recommend to use the baptized in fire Random.jl[98].

[98] https://docs.julialang.org/en/v1/stdlib/Random/

Birthday

In this chapter I used the following libraries.

```
import Random as Rnd # internal library
import Statistics as St # internal library
```

Still, once you read the problem description you may decide to do otherwise.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[99](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[100].

Problem

The following is a classical birthday paradox[101]problem slightly modified by me.

Imagine you're throwing a party for your birthday. You invited 30 people. And now you begin to wonder:

1) what is the probability[102]that any two of your guests were born on the same day of a year?
2) what is the probability[103]that any of your guests were born on the same day of a year that you were born on?

Use Julia to answer the before-mentioned questions for the number of guests, let's say, in the range [4-30].

[99]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/birthday
[100]https://docs.julialang.org/en/v1/stdlib/Pkg/
[101]https://en.wikipedia.org/wiki/Birthday_problem
[102]https://en.wikipedia.org/wiki/Probability
[103]https://en.wikipedia.org/wiki/Probability

Solution

The following solution is based on a computer simulation and relies on a few assumptions, namely:

1) there are always exactly 365 days in a year (no leap years)
2) a person is equally likely to have been born on any day of a year (no seasonal, yearly and other patterns, etc.)
3) the birth date of one person does not influence the birth date of another person (not in twins, siblings, etc.)

None of the above is exactly true, still, those are some reasonable assumptions that will allow us to knack the problem.

OK, let's begin.

```
import Random as Rnd

function getBdaysAtParty(nPeople::Int)::Vec{Int}
    @assert 4 <= nPeople <= 365 "nPeople must be in range [4-365]"
    return Rnd.rand(1:365, nPeople)
end

function getCounts(v::Vector{T})::Dict{T,Int} where T
    counts::Dict{T,Int} = Dict()
    for elt in v
        counts[elt] = get(counts, elt, 0) + 1
    end
    return counts
end
```

We start by defining two simple functions. `getBdaysAtParty` returns a random set of birthdays (`Vec{Int}`) drawn from all the possible days in a year (`1:365`, with possible repetitions). On the other hand, `getCounts` is a function that I copied-pasted from my previous book[104]. It does what it promises, i.e. it returns a summary statistics (`counts`) that tells us how many times a given day in the birthdays (result of `getBdaysAtParty`) appears.

Time to find a way to determine does the event that we are looking for occurred during the party.

[104] https://b-lukaszuk.github.io/RJ_BS_eng/statistics_prob_theor_practice.html

```
function nSameBdays(counts::Dict{Int, Int}, n::Int=2)::Bool
    @assert 2 <= n <= 365 "n must be in range [2-365]"
    return isnothing(findfirst((>=)(n), counts)) ? false : true
end

function anyMyBday(counts::Dict{Int, Int}, myBday::Int=1)::Bool
    @assert 1 <= myBday <= 365 "myBDay must be in range [1-365]"
    return haskey(counts, myBday)
end
```

First, we check for `nSameBdays` with the build in findfirst[105]. That last function accepts a predicate and a collection (like a vector or a dictionary). The predicate is a function that takes an element of a collection (for a dictionary one of its values) as an input and returns a `Bool`. `findfirst` brings back the first index (for vectors) or the first key (for dictionaries) for which `predicate(vector's element)` or `predicate(dictionary's value)` is `true`, respectively. Often a predicate is just an anonymous function[106], which in our case could be `key -> key >= n`. Instead, here I used a partial function application[107]. The `(>=)(n)` is an equivalent of the `>= n` mentioned in the previous sentence. It is just a function applied to one argument. The function still lacks an argument on its left site, just before the `>`. This missing argument will be any element of our collection that is currently being tested. Anyway, one more point to notice, if `findfirst` finds no element for which the predicate is `true` then it returns nothing[108]. Therefore, to get our final answer we send the possible result through isnothing[109]and a ternary expression[110]. Technically, we could have shorten the last line to `return !isnothing(firndfirst((>=)(n), counts))` (or even without the return), but I thought that this might be too much condensation for a one line of code.

[105] https://docs.julialang.org/en/v1/base/arrays/#Base.findfirst-Tuple%7BFunction,%20Any%7D
[106] https://docs.julialang.org/en/v1/manual/functions/#man-anonymous-functions
[107] https://bkamins.github.io/julialang/2024/02/23/fix.html
[108] https://docs.julialang.org/en/v1/base/constants/#Core.nothing
[109] https://docs.julialang.org/en/v1/base/base/#Base.isnothing
[110] https://b-lukaszuk.github.io/RJ_BS_eng/julia_language_decision_making.html#sec:ternary_expression

Our second function, `anyMyBday`, is much simpler. It just uses `haskey` to check if our day of birth occurred in the birthdays of the guests at our party (`counts`). Fun, fact, under our assumptions, it does not matter whether you were born on the first or 365th day of the year. Go ahead, change the default value to your actual day of birth and compare the result with the one provided later in this chapter.

OK, time to estimate the probability of success, i.e. the ratio between the number of times the event took place to the total trials (number of simulations).

```
# isEventFn(Dict{Int, Int}) -> Bool
function getProbSuccess(nPeople::Int, isEventFn::Function,
                        nSimulations::Int=100_000)::Flt
    @assert 4 <= nPeople <= 366 "nPeople must be in range [4-365]"
    @assert 1e4 <= nSimulations <= 1e6 "nSimulations not in range
[1e4-1e6]"
    successes::Vec{Bool} = Vec{Bool}(undef, nSimulations)
    for i in 1:nSimulations
        peopleBdays = getBdaysAtParty(nPeople)
        counts = getCounts(peopleBdays)
        eventOccured = isEventFn(counts)
        successes[i] = eventOccured
    end
    return sum(successes)/nSimulations
end
```

First, we initialize the vector that will hold the results of our simulations (`successes`). The `Vec{Bool}(undef, nSimulations)` creates a vector of size specified in `nSimulations` that is currently occupied by some unspecified (`undef` - undefined) values (basically garbage that is currently in a specific place of our computer's memory). We will fill the `successes` with the values of interest in the for loop. For each simulation, we throw a party and get the guests' birthdays (`peopleBdays`). We obtain `counts` for them and test did the event that we consider a success occurred that time (with `isEventFn`). We place the occurrence into the proper spot (`[i]`) of our vector of `successes`. Notice, that here `peopleBdays`, `counts`, and `eventOccured` are local (helper) variables that are visible only in the for loop. Anyway, all that's left to do is to calculate the probability. The `sum`

function treats any `true` as `1` and `false` as 0, hence it returns the number of successes, which we divide by the number of simulations.

Let's see how it works.

```
Rnd.seed!(101)
peopleAtParty = 5:30
probsAnySameBdays = [getProbSuccess(n, nSameBdays) for n in
peopleAtParty]
probsMyBday = [getProbSuccess(n, anyMyBday) for n in peopleAtParty]
```

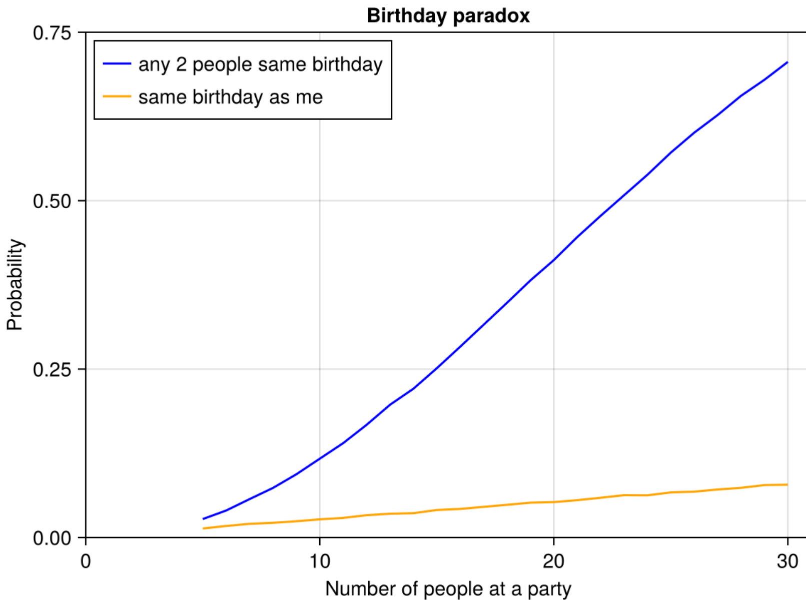


Figure 2: Birthday paradox probabilities.

So, with 23 people at a party the probability that any 2 of them have a birthday on the same day is roughly 50% or 0.5. The probability, that any person was born the same day that I have been born is around 8% or 0.08 for 30 people at a party. Actually, this last probability is fairly straightforward to calculate. For the reasons explained here[111]it is just

[111]https://b-lukaszuk.github.io/RJ_BS_eng/statistics_intro_probability_properties.html

$\frac{1}{365} * n\ people\ at\ party$ so our mean absolute error[112] is roughly equal to:

```
import Statistics as St

probsMyBdayTheor = (1/365) .* peopleAtParty
(probsMyBday .- probsMyBdayTheor) .|> abs |> St.mean
```

0.0013415015806111702

which isn't all that bad.

As a mini-exercise you may tweak the code a little and estimate the probability that any 3 people at a party were born on the same day.

112 https://en.wikipedia.org/wiki/Mean_absolute_error

Logo

In this chapter I used the following libraries.

```
import CairoMakie as Cmk # external library
import Random as Rnd # internal library
```

Still, once you read the problem description you may decide to do otherwise.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[113](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[114].

Problem

Julia is a nice programming language with over 10'000 registered packages for community use. Some of them come with cool logos.

In this task your job is to use your favorite plotting library to reproduce the logo of JuliaStats[115]that you may find here[116](it doesn't have to be exact).

Solution

The logo is composed of three disks build of points (scatter-plot) of three colors: red, green and purple. So let's start small and try to replicate it by placing the points (only three for now) on a graph in their correct locations.

```
import CairoMakie as Cmk

function drawLogo()::Cmk.Figure
    centersXs::Vec{Flt} = [1, 5.5, 10]
```

[113]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/logo
[114]https://docs.julialang.org/en/v1/stdlib/Pkg/
[115]https://juliastats.org/
[116]https://juliastats.org/images/logo.png

```
    centersYs::Vec{Flt} = [1, 6, 1]
    colors::Vec{Str} = ["red", "green", "purple"]
    fig::Cmk.Figure = Cmk.Figure()
    ax::Cmk.Axis = Cmk.Axis(fig[1, 1])
    Cmk.scatter!(ax, centersXs, centersYs, color=colors, markersize=50)
    return fig
end
```

> ***Note:*** `CairoMakie` uses `Colors.jl`, the list of available color names is to be found here[117].

The function is pretty simple. First, we defined the locations (based on a guess and subsequent try and error) of the points with respect to the x- (`centersXs`) and y-axis (`centersYs`), as well as their colors. Next we added the points (`scatter!`) to the axis object (`ax`) attached to the figure object (`fig`).

Time to take a look.

```
drawLogo()
```

[117] https://juliagraphics.github.io/Colors.jl/stable/namedcolors/

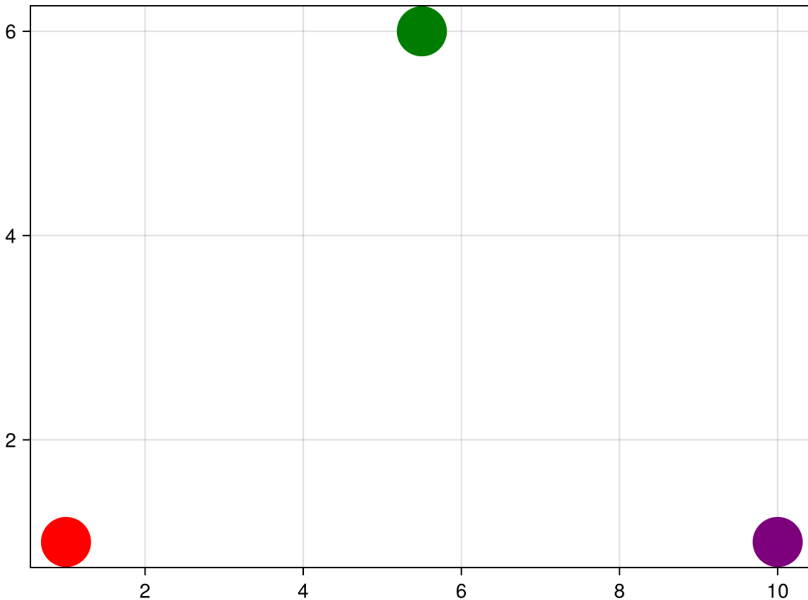


Figure 3: Replicating JuliaStats logo. Attempt 1.

So far, so good. Now instead of a one big point we will need a few thousand smaller points (let's say `markersize=10`) concentrated around the center of a given group. Moreover, the points should be partially transparent [to that end we will use `alpha` keyword argument (`alpha=0` - fully transparent, `alpha=1` - fully opaque)]. One more thing, the points need to be randomly scattered, but with a greater density closer to the group's center. This last issue will be solved by obtaining random numbers from the normal distribution[118]with the mean equal 0 and the standard deviation equal 1. This is what `randn` function will do. Thanks to this roughly 68% of the numbers will be in the range −1 to 1, 95% in the range −2 to 2, and 99.7% in the range −3 to 3 (greater density around the mean). Behold.

```
import Random as Rnd

function drawLogo()::Cmk.Figure
```

[118]https://b-lukaszuk.github.io/RJ_BS_eng/statistics_normal_distribution.html

```
    centersXs::Vec{Flt} = [1, 5.5, 10]
    centersYs::Vec{Flt} = [1, 6, 1]
    colors::Vec{Str} = ["red", "green", "purple"]
    numOfPoints::Int = 3000
    fig::Cmk.Figure = Cmk.Figure()
    ax::Cmk.Axis = Cmk.Axis(fig[1, 1])
    for i in eachindex(colors)
        Cmk.scatter!(ax,
                     centersXs[i] .+ Rnd.randn(numOfPoints),
                     centersYs[i] .+ Rnd.randn(numOfPoints),
                     color=colors[i], markersize=10, alpha=0.25)
    end
    return fig
end
```

Let's test it.

```
Rnd.seed!(303) # needed to make it reproducible
drawLogo()
```

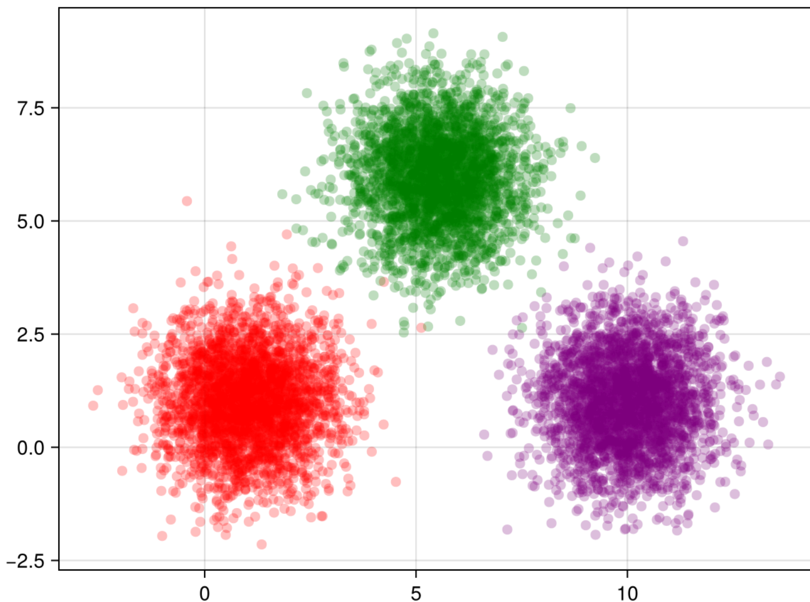


Figure 4: Replicating JuliaStats logo. Attempt 2.

We're almost there. Notice, that `CairoMakie` nicely centered the plot, so we don't need to do this ourselves manually.

Anyway, all that's left to do is to remove the unnecessary elements from the picture. A brief look into the documentation of the package indicates that hidedecorations and hidespines[119] should do the trick.

```
function drawLogo()::Cmk.Figure
    centersXs::Vec{Flt} = [1, 5.5, 10]
    centersYs::Vec{Flt} = [1, 6, 1]
    colors::Vec{Str} = ["red", "green", "purple"]
    numOfPoints::Int = 3000
    fig::Cmk.Figure = Cmk.Figure()
    ax::Cmk.Axis = Cmk.Axis(fig[1, 1])
    Cmk.hidedecorations!(ax)
    Cmk.hidespines!(ax)
    for i in eachindex(colors)
        Cmk.scatter!(ax,
                     centersXs[i] .+ Rnd.randn(numOfPoints),
                     centersYs[i] .+ Rnd.randn(numOfPoints),
                     color=colors[i], markersize=10, alpha=0.25)
    end
    return fig
end
```

And voila.

```
Rnd.seed!(303) # needed to make it reproducible
drawLogo()
```

[119] https://docs.makie.org/v0.21/reference/blocks/axis#Hiding-Axis-spines-and-decorations

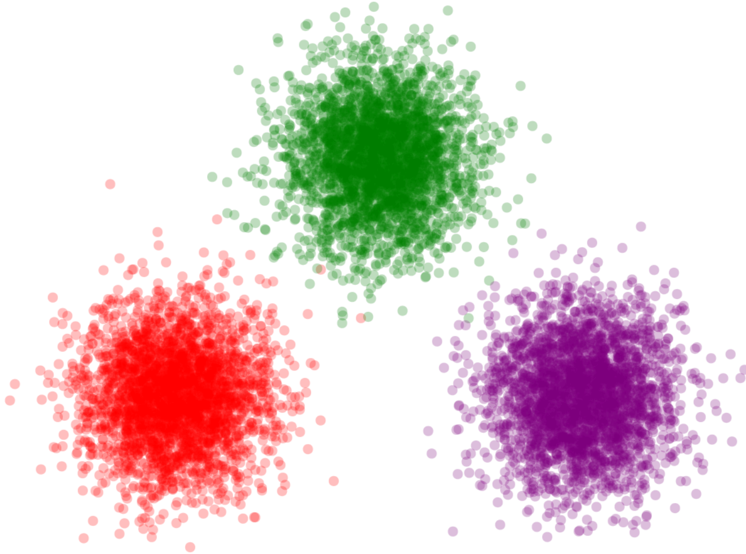


Figure 5: Replicating JuliaStats logo. Attempt 3.

We obtained a decent replica of the original. Of course, kudos go to the authors of the original image for their creativity and artistic taste.

Binary

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[120](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[121].

Problem

Numbers, or information in general, can be written down in different forms. The most basic one, and the only one that is actually understood by a computer, is a binary. Usually, it is depicted by a sequence of 1s and 0s, but it could be anything really. Anything, that can take two separate states. The once common CDs[122]or DVDs[123]are a sequence of laser burns (tiny pits) in a spiral track (burn - 1, no burn - 0). The hard disk drives[124]store data as magnetized spots, whereas SSD drives[125]cash it as electrons trapped in tiny transistors. The data on a hard drive can be interpreted as decimal digits, which in turn can be interpreted as characters from an alphabet (like ASCII[126]). Moreover, the data can be read into memory (RAM) and send to a processor for calculations.

This time you task is to read about the binary numbers[127]. Alternatively, you may watch some online videos (e.g. from Khan

[120]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/binary
[121]https://docs.julialang.org/en/v1/stdlib/Pkg/
[122]https://en.wikipedia.org/wiki/Compact_disc
[123]https://en.wikipedia.org/wiki/DVD
[124]https://en.wikipedia.org/wiki/Hard_disk_drive
[125]https://en.wikipedia.org/wiki/Solid-state_drive
[126]https://en.wikipedia.org/wiki/ASCII
[127]https://en.wikipedia.org/wiki/Binary_number

Academy[128]) on the topic. Next, to get a better grasp of the subject write:

- a function that transforms a binary number to its decimal counterpart
- a function that transforms a decimal number to its binary counterpart
- a function that adds two binary numbers
- a function that multiplies two binary numbers

For simplicity, assume that the functions will operate only on natural numbers in the range of let's say 0 to 1024 (in decimal). You may check your functions against the built-in `string` and `parse` functions. For instance, `string(3, base=2)` converts the decimal (3) to its binary representation and `parse(Int, "11", base=2)` performs the opposite action.

Solution

Let's start with our `dec2bin` converter, we'll base it on the Khan Academy videos[129]and similarities with the decimal number system.

The decimal system[130]operates on base 10. A number, let's say: one hundred and twenty-three (123), can be written with digits ([0-9]) placed in three slots. We know this because three slots allow us to write $10^3 = 1000$ numbers ([0-999]), whereas two slots are good only for $10^2 = 100$ numbers ([0-99], compare also with the exercise 1[131]and its solution[132]). The number 123 is actually a sum of one hundred, two tens, and three units ($1*100 + 2*10 + 3*1$ = `sum([1*100, 2*10, 3*1])` = 123). Equivalently, this can be written with the consecutive powers of ten (10^x = `10^x` in Julia's code), i.e. $1*10^2 + 2*10^1 + 3*10^0$

[128] https://www.youtube.com/watch?v=ku4KOFQ-bB4&list=PLS---sZ5WJJvsjaAQZKwTwxl910xUdO98

[129] https://www.youtube.com/watch?v=ku4KOFQ-bB4&list=PLS---sZ5WJJvsjaAQZKwTwxl910xUdO98

[130] https://en.wikipedia.org/wiki/Decimal

[131] https://b-lukaszuk.github.io/RJ_BS_eng/statistics_intro_exercises.html#sec:statistics_intro_exercise1

[132] https://b-lukaszuk.github.io/RJ_BS_eng/statistics_intro_exercises_solutions.html#sec:statistics_intro_exercise1_solution

= `sum([1*10^2, 2*10^1, 3*10^0])` = 123. Pause for a moment and make sure you got that.

Similar reasoning applies to binary numbers, but here we operate on the powers of two. A decimal number, let's say: fourteen (14), can be written with digits ([0-1]) placed in four slots. This is because 2^4 allows us to write down 16 numbers (in binary: [0000-1111], in decimal: [0-15]), whereas 2^3 suffices only for 8 numbers (in binary: [000-111], in decimal: [0-7]). Each slot (from right to left) represents subsequent powers of two, i.e. ones ($2^0 = 1$), twos ($2^1 = 2$), fours ($2^2 = 4$), and eights ($2^3 = 8$). Once again we sum the digits to get our encoded number `1110` = $1*2^3 + 1*2^2 + 1*2^1 + 1*2^0$ = $1*8 + 1*4 + 1*2 + 1*0$ = `sum([1*8, 1*4, 1*2, 1*0])` = 14. Time to put that knowledge to good use by writing our `dec2bin`.

```
function getNumOfBits2codeDec(dec::Int)::Int
    @assert 0 <= dec <= 1024 "dec must be in range [0-1024]"
    dec = (dec == 0) ? 1 : dec
    for i in 1:dec
        if 2^i > dec
            return i
        end
    end
    return 0 # should never happen
end

function dec2bin(dec::Int)::Str
    @assert 0 <= dec <= 1024 "dec must be in range [0-1024]"
    nBits::Int = getNumOfBits2codeDec(dec)
    result::Vec{Char} = fill('0', nBits)
    bitDec::Int = 2^(nBits-1) # -1, because powers of 2 start at 0 not 1
    for i in eachindex(result)
        if bitDec <= dec
            result[i] = '1'
            dec -= bitDec
        end
        bitDec = div(bitDec, 2)
    end
    return join(result)
end
```

We begin by declaring `getNumOfBits2codeDec`, a function that will help us to find how many bits (slots) we need in order to code a given decimal as a binary number. It does so by using a 'brute force'

approach (`for i in 1:dec`) with an early stop mechanism (`if 2^i > dec`). As an alternative we may consider to use the built-in `log2` function, but the approach presented here just seemed so natural and in line with the previous explanations that I just couldn't resist.

As for the `dec2bin` function we start by declaring a few helper variables: `nBits` - number of bits/slots we need to fill in order to code our number, `result` - a vector that will hold our final binary representation, and `bitDec` - a variable that will store the consecutive powers of 2 (expressed in decimal) coded by a given bit. Next, we traverse the bits/slots of our `result` from left to right. When the current power of two is smaller or equal to the decimal we got to encode (`if bitDec <= dec`) then we set that particular bit to 1 (`result[i] = '1'`) and reduce the decimal we still need to encode by that value (`dec -= bitDec`). Moreover, we update (reduce) our consecutive powers of two (`bitDec`) by using the 2 divisor (`div(bitDec, 2)`). `div` performs an integer division (that drops the decimal part of the quotient). We use it because in the array of the powers of two: $2^4 = 16, 2^3 = 8, 2^2 = 4, 2^1 = 2, 2^0 = 1$ each number gets two times smaller. The `div(bitDec, 2)` protects us from the case when the integer division by 2 with standard `bitDec/2` would not be possible (in the last turnover of the loop `bitDec` will be equal to `1`).

Let's see how it works with a couple of numbers.

```
lpad.(dec2bin.(0:8), 4, '0')
```

```
[
"0000",
"0001",
"0010",
"0011",
"0100",
"0101",
"0110",
"0111",
"1000"
]
```

Time for a benchmark against the built-in `string` function.

```
all([dec2bin(i) == string(i, base=2) for i in 0:1024]) # Python like
# or simply, more Julia style
dec2bin.(0:1024) == string.(0:1024, base=2)
```

true

It appears we did just fine as the function produces the same results as `string`.

Once we got it reversing the process should be a breeze.

```
function isBin(bin::Char)::Bool
    return bin in ['0', '1']
end

function isBin(bin::Str)::Bool
    return isBin.(collect(bin)) |> all
end

function bin2dec(bin::Str)::Int
    @assert isBin(bin) "bin is not a binary number"
    pwr::Int = length(bin) - 1 # -1, because powers of 2 start at 0 not
1
    result::Int = 0
    for b in bin
        result += (b == '1') ? 2^pwr : 0
        pwr -= 1
    end
    return result
end
```

Once again, we we start our main function (`bin2dec`) with the definition of a few helper variables: `pwr` - which holds a power of the current bit which is in the range from `length(bin)-1` to `0` (from the leftmost to the rightmost bit), and `result` which is just a sum of all bits expressed in decimal system. We build the sum (`result +=`) bit by bit (`for b in bin`), but only for the bits equal `'1'` (`(b ==1) ? 2^pwr`) while reducing the power for the next bit as we shift right (`pwr -= 1`). Finally, all that's left to do is to `return` the `result`.

Let's see how we did.

```
lpad.(dec2bin.(0:8), 4, '0') .|> bin2dec
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

It looks good, and now for a more comprehensive benchmark.

```
bin2dec.(string.(0:1024, base=2)) == 0:1024
```

true

Again, it seems that we can't complain.

OK, time to add two binaries.

> ***Note:*** Before you move further, I suggest you take a pen and paper and do the addition and multiplication for some decimals and binaries to get a better grasp of the process that we will translate into Julia's code.

```
# returns (carried bit, running bit)
function add(bin1::Char, bin2::Char)::Tuple{Char, Char}
    @assert isBin(bin1) && isBin(bin2) "both inputs must be binary bits"
    if bin1 == '1' && bin2 == '1'
        return ('1', '0')
    elseif bin1 == '0' && bin2 == '0'
        return ('0', '0')
    else
        return ('0', '1')
    end
end

function getEqlLenBins(bin1::Str, bin2::Str)::Tuple{Str, Str}
    if length(bin1) >= length(bin2)
        return (bin1, lpad(bin2, length(bin1), '0'))
    else
        return getEqlLenBins(bin2, bin1)
    end
end
```

In order to add two binary numbers we need to define how to add two binary digits (`bin1::Char` and `bin2::Char`) first. Just like in the decimal system we need to handle the overflow, e.g. when we add $26 + 8$, first we add 8 and 6 to get 14. Four (the second digit of 14) becomes the running bit and 1 (the first digit of 14) becomes a carried

bit that we add to 2 (the first digit in 26) to get our final score which is 34. By analogy, `add(bin1::Char, bin2::Char)` returns a tuple (`Tuple{Int, Int}`) with the carried and running bit, respectively.

Additionally, we defined `getEqlLenBins` that makes sure the two numbers (`bin1` and `bin2`) are of equal length. This is because the upcoming `add(bin1::Str, bin2::Str)` will rely on the above `add(bin1::Char, bin2::Char)`, that's why the shorter number will be padded on the left with zeros (by `lpad`), which is a neutral digit in addition (in decimal adding 26+8 is the same as adding 26+08). Two points of notice. First of all, make sure to use >= and not > in the first `if` statement otherwise you will end up with an infinite recursion (see Section 10) and stack overflow[133]error in some cases (test yourself and explain why it will happen for `getEqlLenBins("01", "10")`). Secondly, the recursive call `getEqlLenBins(bin2, bin1)` effectively swaps the numbers. This is a neat trick and makes no difference here (since both addition and multiplication are commutative[134]), but may cause troubles otherwise. Anyway, time for `add(bin1::Str, bin2::Str)`.

```
function isZero(bin::Char)::Bool
    return bin == '0'
end

function isZero(bin::Str)::Bool
    return isZero.(collect(bin)) |> all
end

function add(bin1::Str, bin2::Str)::Str
    @assert isBin(bin1) && isBin(bin2) "both inputs must be binary
numbers"
    bin1, bin2 = getEqlLenBins(bin1, bin2)
    carriedBit::Char = '0'
    runningBit::Char = '0'
    runningBits::Str = ""
    carriedBits::Str = "0"
    for (b1, b2) in zip(reverse(bin1), reverse(bin2))
        carriedBit, runningBit = add(b1, b2)
        runningBits = runningBit * runningBits
```

[133]https://en.wikipedia.org/wiki/Stack_overflow

[134]https://en.wikipedia.org/wiki/Commutative_property

```
        carriedBits = carriedBit * carriedBits
    end
    if isZero(carriedBits)
        return isZero(runningBits) ? "0" : lstrip(isZero, runningBits)
    else
        return add(runningBits, carriedBits)
    end
end
```

The key function is rather simple. First, we align the binaries to contain the same number of bits/slots (`getEqlLenBins`) and declare (and initialize) a few helper variables. Next, we move from right to left (`reverse` functions) by the corresponding bits (`b1`, `b2`) of our binary numbers (`bin1` and `bin2`). We add the bits together (`add(b1, b2)`) and prepend (`*` - glues `Chars` and `Strings` together) the obtained `runningBit` and `carriedBit` to `runningBits` and `carriedBits`, respectively. Once we traveled every bit of `bin1` and `bin2` (thanks to the `for` loop). We check if the `carriedBits` is equal to zero (i.e. all bits are equal `0`). If so (`if isZero(carriedBits)`), then we `return` our `runningBits` but without the excessive zeros (`lstrip(isZero, runningBits)`) that might have been produced on the left site (since e.g. the binary `010` is the same as `10`, just like the decimal `03` is the same as `3`). However, if `runningBits` is equal zero (`isZero(runningBits) ?`) we return `"0"` (because in this case `lstrip` would have returned an empty string, i.e. `""`). Otherwise (`else`), we just add the carried bits to the running bits (recursive `add(runningBits, carriedBits)` call). Notice, that in order for the last statement to work `carriiedBits` need to be moved by 1 to the left with respect to the `runningBits`. That is why, we initialized them with `"0"` and not an empty string `""` in the first place (`carriedBits::Str = "0"`). If the last two sentences are not clear, then go ahead take a pen and paper and add two simple decimals with the carry (like in the primary school). Then you will see that the carried bit is moved to the left.

Some test would be in order. And here is our testing powerhouse.

```
# binFn(Str, Str) -> Str, decFn(Int, Int) -> Int
function doesBinFnWork(dec1::Int, dec2::Int,
                       binFn::Function, decFn::Function)::Bool
    bin1::Str = binFn(dec2bin(dec1), dec2bin(dec2))
    bin2::Str = string(decFn(dec1, dec2), base=2)
    return bin1 == bin2
end
```

`doesBinFnWork` accepts two decimals (`dec1`, `dec2`) and two functions (`binFn`, `decFn`) as its arguments. Each of the functions should accept two arguments (`binFn` binaries, and `decFn` decimals) and perform the same operation on them. Notice, that inside of `doesBinFnWork` both `dec1` and `dec2` are converted to binaries and send to `binFn`, whereas the result of `decFn(dec1, dec2)` is converted to binary. In the end `bin1` and `bin2` are compared to make sure that they are equal. All set, time for a test.

```
# running this test may take a few seconds (513x513 matrix)
tests = [doesBinFnWork(a, b, add, +) for a in 0:512, b in 0:512]
all(tests)
```

true

Another day, another dollar. Time for the multiplication.

```
function multiply(bin1::Char, bin2::Char)::Char
    @assert isBin(bin1) && isBin(bin2) "both inputs must be binary bits"
    if bin1 == '1' && bin2 == '1'
        return '1'
    else
        return '0'
    end
end

function multiply(bin1::Str, bin2::Str)::Str
    @assert isBin(bin1) && isBin(bin2) "both inputs must be binary
numbers"
    total::Str = "0"
    curProd::Str = "0"
    nZerosToPad::Int = 0
    for b in reverse(bin2)
        curProd = multiply.(b, collect(bin1)) |> join
        total = add(total, curProd * '0'^nZerosToPad)
        nZerosToPad += 1
```

```
    end
    return total
end
```

Again, we begin by defining how to multiply two individual bits, and again it resembles the multiplication in the decimal system. Once we got it, we move to multiply the whole numbers (`multiply(bin1::Str, bin2::Str)`). Just like in the decimal system we multiply all the bits (from right to left) from the second number (`for b in reverse(bin2)`) by the bits in the first number (`multiply.(b, collect(bin1)) |> join`). After each multiplication the product (`curProd`) of `b` times `bin1` is summed (`add(total, curProd * '0'^nZerosToPad)`) into a grand `total`. Just like in the decimal system, `curProd` is shifted to left every time we do that, which is why we defined and increased `nZerosToPad` variable. Let's test it out.

```
# 33x33 matrix so it should be relatively fast
tests = [doesBinFnWork(a, b, multiply, *) for a in 0:32, b in 0:32]
all(tests)
```

true

I guess we did it again. Good for us.

The Answer

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[135](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[136].

Problem

In one of the novels of Douglas Adams a supercomputer was asked to provide the Answer to the Ultimate Question of Life, the Universe, and Everything[137]. The answer turned out to be 42. At first it seems to make no sense until you realize that the ultimate question was: "What do you get if you multiply six by nine?".

So here is a task for you: modify `dec2bin` function from Section 14 and name it, e.g. `dec2baseN`. Use the latter to test numerical systems of the base 2 (binary[138]) to 16 (hexadecimal[139]) and see which numbers multiplied by each other give 42. Check if both the question and the answer make sense in any of the different positional number systems[140].

Solution

OK, we'll start by defining a few constants that will be useful later on.

---

[135] https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/the_answer

[136] https://docs.julialang.org/en/v1/stdlib/Pkg/

[137] https://en.wikipedia.org/wiki/Phrases_from_The_Hitchhiker%27s_Guide_to_the_Galaxy#The_Answer_to_the_Ultimate_Question_of_Life,_the_Universe,_and_Everything_is_42

[138] https://en.wikipedia.org/wiki/Binary_number

[139] https://en.wikipedia.org/wiki/Hexadecimal

[140] https://en.wikipedia.org/wiki/Positional_notation

```
const CHARS = vcat('0':'9', 'a':'f') # vcat - vector concatenate
const MIN_BASE = 2
const MAX_BASE = 16
```

`CHARS` contains symbols used to denote numbers in different positional number systems. In the binary numeral system (`MIN_BASE = 2`) we use the first two characters (`CHARS[1:2]`, i.e. `'0' and '1'`) to code a number. In the base-3 system we use the first three characters, i.e. `CHARS[1:3]`, i.e. `'0', '1', '2'`. Next, the first four, five, six and so forth characters up until the hexadecimal numeral system (`MAX_BASE = 16`) where we use them all.

Now, for the converter:

```
function dec2baseN(dec::Int, n::Int)::Str
    @assert MIN_BASE <= n <= MAX_BASE "n not in [2 - 16]"
    @assert 0 <= dec <= 1024 "dec must be in range [0-1024]"
    result::Str = ""
    while dec != 0
        # (dec % n) - C-like indexing [0 - (n-1)]
        result *= CHARS[(dec % n) + 1]
        dec = div(dec, n)
    end
    return isempty(result) ? "0" : reverse(result)
end
```

Here, we used an algorithm that differs from to the one in Section 14.2 . Basically, we divide a decimal number (`dec`) by the base `n` in which it is to be coded. Our algorithm comes from the Wikipedia[141]and although it was designed for a decimal to binary conversion it works also for other numerical systems. The page in the link states:

> To convert from a base-10 integer to its base-2 (binary) equivalent, the number is divided by two. The remainder is the least-significant bit. The quotient is again divided by two; its remainder becomes the next least significant bit.

[141]https://en.wikipedia.org/wiki/Binary_number#Decimal_to_binary

Of course, instead of 2 we used a reminder of `n` and the quotient divided by `n` (`div(dec, n)`). Of note the modulo operator (`%`) returns the reminder in the range `[0 - (n-1)]`, e.g.

```
[i % 3 for i in 0:6]
```

```
[0, 1, 2, 0, 1, 2, 0]
```

Therefore, in order to convert it to Julia's indexing system we had to add `+1`. Moreover, during the turnovers of our `while` loop the least significant reminder was appended (`*=`) to the `result`, then the second least significant, then the third, etc. Due to this process the least significant slot was on the left side of the string (`result`), whereas the most important slot was on the right side of it. Hence, we `reversed` the `result` in our last step.

Let's compare its action against Julia's built-ins (the `string` function):

```
[dec2baseN(i, b) == string(i, base=b)
    for b in MIN_BASE:MAX_BASE for i in 0:1024] |> all
```

true

Nothing to complain about. And now for the answer:

```
for base in MIN_BASE:MAX_BASE
    for dec1 in 0:(base-1), dec2 in 0:(base-1) # 1 digit nums only
        n1 = dec2baseN(dec1, base)
        n2 = dec2baseN(dec2, base)
        product = dec2baseN(dec1 * dec2, base)
        if product == "42"
            println("base $base: $n1 * $n2 = $product")
        end
    end
end
```

```
base 7: 5 * 6 = 42
base 7: 6 * 5 = 42
base 10: 6 * 7 = 42
base 10: 7 * 6 = 42
base 12: 5 * a = 42
```

```
base 12: a * 5 = 42
base 13: 6 * 9 = 42
base 13: 9 * 6 = 42
base 16: 6 * b = 42
base 16: b * 6 = 42
```

So it seems that by multiplying the above two single digit numbers in bases: 7, 10, 12, 13, and 16 we get 42 as a result. Of the above only base 13 fulfills the criteria of both the question and the answer.

If you feel that doing the multiplication in decimal and translating it to `product` in the desired base system was like cheating, then you may try to re-implement `add` and `multiply` functions from Section 14.2 . A natural way to do that would be to create an addition table for `add(num1::Char, num2::Char, base::Int)::Tuple{Char, Char}` and a multiplication table for `multiply(num1::Char, num2::Char, base::Int)::Tuple{Char, Char}` to use. Thanks to them the functions would rely on the same process as longhand pen and paper calculations. Unfortunately, nobody taught us addition and multiplication tables in numerical systems of other bases when we were in school. And so, again, the easiest way to create such tables would be to use conversion from decimals. For that reason, here I will stay with the solution presented above. Still, you may check the code snippets for this chapter[142]as they contain the other method (or better try to implement it yourself).

[142]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/the_answer

The Doors

In this chapter I used the following libraries.

```
import DataFrames as Dfs # external library
import Random as Rnd # internal library
```

Still, once you read the problem description you may decide to do otherwise.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[143](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[144].

Problem

There is this famous game-show host problem, also known as the Monty Hall problem[145]that goes something like this.

Imagine there is a game-show in which a person may win a new car. The car is hidden behind one of three doors. The host chooses a player (called a trader) from the people in the studio and asks them to pick one door. The choice is Door 1. The host knows what's behind all the doors. He decided to open Door 3, behind which there's a goat. Now the trader got a choice: stay with their initial choice, or switch to still unopened Door 2.

Imagine you are the trader. What would you do? Is it in your interest to make a switch? Use Julia to figure it out.

Solution

One way to answer the question is to use the so called Bayes's

[143]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/the_doors

[144]https://docs.julialang.org/en/v1/stdlib/Pkg/

[145]https://en.wikipedia.org/wiki/Monty_Hall_problem

Theorem, as explained by Allen B. Downey here[146].

> ***Note:*** Below, you will find a shortened explanation. To understand the topic more satisfactorily you will likely need to read the first two chapters of Think Bayes[147] by Allen B. Downey.

First the theorem:

$$P(A \mid B) = \frac{P(A) * P(B \mid A)}{P(B)}$$

which can be also written as:

$$P(H \mid D) = \frac{P(H) * P(D \mid H)}{P(D)} \tag{9}$$

where:

- P(H|D) - probability of a hypothesis given the obtained data (aka **posterior**),
- P(H) - probability of a given hypothesis upfront (aka **prior**),
- P(D|H) - probability of obtaining such data given the discussed hypothesis is true (aka **likelihood**),
- P(D) - the total probability of the data under all the considered hypotheses.

Regarding the **priors** or P(H), initially it is equally likely for the car to be behind any of the three doors. Therefore, the probability that the car is behind Door X is: 1 out of 3, so 1/3. This can be represented as:

```
import DataFrames as Dfs

df = Dfs.DataFrame(Dict("Door" => 1:3))
df.prior = repeat([1//3], 3) # 1//3 is Rational (a fraction)
df
```

[146] https://allendowney.github.io/ThinkBayes2/chap02.html#the-monty-hall-problem

[147] https://allendowney.github.io/ThinkBayes2/index.html

Door	prior
1	1//3
2	1//3
3	1//3

Table 1: Table 1: The doors. Priors.

Let's move to **likelihood** or P(D|H). If the car was behind Door 1, then the host may open either Door 2 or Door 3. Hence, the probability of him opening Door 3 (as he did in the problem description) is 1 out of 2, so 1/2 or 0.5. If the car was behind Door 2, then the host had no other option but to open Door 3 (since Door 1 is the trader's choice and he doesn't want to reveal the car behind Door 2). Therefore, in this case the probability is equal to 1 (certain event). If the car were behind Door 3, then there is no way the host would have opened it (since it would have revealed the car and spoiled the game). So, in this case the probability is 0.

Let's add this information to the table.

```
df.likelihood = [1//2, 1, 0]
df
```

Door	prior	likelihood
1	1//3	1//2
2	1//3	1
3	1//3	0//1

Table 2: Table 2: The doors. Likelihoods.

Now we perform a so called Bayesian update, e.g. per the formula in Equation 9 : first we multiply $P(H)$ (prior) by $P(D \mid H)$ (likelihood) then we divide it by $P(D)$ (sum of all probabilities) like so:

```
function bayesUpdate!(df::Dfs.DataFrame)::Nothing
    unnorm = df.prior .* df.likelihood
    df.posterior = unnorm ./ sum(unnorm)
    return nothing
```

```
end

bayesUpdate!(df)
df # to see Rationals as floats: convert.(Flt, df.posterior)
```

Door	prior	likelihood	posterior
1	1//3	1//2	1//3
2	1//3	1	2//3
3	1//3	0//1	0//1

Table 3: Table 3: The doors. Posteriors.

Clearly, switching to Door 2 gives the trader a better chance of winning the car ($P = \frac{2}{3} \approx 0.66$) than remaining with the original choice ($P = \frac{1}{3} \approx 0.33$).

> ***Note:*** To see the posterior as floats you may use, e.g. `convert.(Flt, df.posterior)`.

If that doesn't convince you then let's do a computer simulation.

```
import Random as Rnd

mutable struct Door
    isCar::Bool
    isChosen::Bool
    isOpen::Bool
end

function get3RandDoors()::Vec{Door}
    cars::Vec{Bool} = Rnd.shuffle([true, false, false])
    chosen::Vec{Bool} = Rnd.shuffle([true, false, false])
    return [Door(car, chosen, false)
            for (car, chosen) in zip(cars, chosen)]
end

# open first non-chosen, non-car door
function openEligibleDoor!(doors::Vec{Door})::Nothing
    @assert length(doors) == 3 "need 3 doors"
    indEligible::Int = findfirst(d -> !d.isCar && !d.isChosen, doors)
    doors[indEligible].isOpen = true
    return nothing
```

```
end

# swap to first non-chosen, non-open door
function swapChoice!(doors::Vec{Door})::Nothing
    @assert length(doors) == 3 "need 3 doors"
    indCurChosen::Int = findfirst(d -> d.isChosen, doors)
    ind2Choose::Int = findfirst(d -> !d.isChosen && !d.isOpen, doors)
    doors[indCurChosen].isChosen = false
    doors[ind2Choose].isChosen = true
    return nothing
end
```

We start by defining a `Door` structure that has all the necessary fields so that we can simulate our game-show. Notice the `mutable` keyword, it will allow us to change a property of a `Door` in-place. Anyway, we follow the structure definition with a random generator of three doors (`get3RandDoors`), a door opener (`openEligibleDoor`) and `swapChoice`. All the above act per the game description. Of note, findfirst[148]accepts a predicate and a vector. The predicate is a function that is executed on consecutive elements of the vector (`doors`) and returns `true` or `false` for each of them. In the end `findfirst` returns the first index from the vector (`doors`) for which the predicate was `true` or `nothing` if no such thing happened. Normally, we should handle both the possible cases (`Int` if `true` and `nothing` if all the tests came negative). However, if we're pretty sure to get the `Int` and actually we want to get the error if we're mistaken, then we may leave it as in the code snippet above.

Now, we only need a way to determine did the player win.

```
function didTraderWin(doors::Vec{Door})::Bool
    indWinner::Union{Nothing, Int} = findfirst(
        d -> d.isChosen && d.isCar, doors)
    return isnothing(indWinner) ? false : true
end

function getResultOfDoorsGame(shouldSwap::Bool=false)::Bool
    doors::Vec{Door} = get3RandDoors()
    openEligibleDoor!(doors)
    if shouldSwap
```

[148]https://docs.julialang.org/en/v1/base/arrays/#Base.findfirst-Tuple%7BFunction,%20Any%7D

```
        swapChoice!(doors)
    end
    return didTraderWin(doors)
end
```

Of note, in the above snippet we used `findfirst` once more. However, this time we may not find the index of a winner (`indWinner`). We mark that possibility with an `Union` type (`indWinner` is an `Int` if behind the doors chosen by the trader is a car or `nothing` otherwise). Lastly, we handle such a situation with isnothing[149]and a ternary expression[150].

Now we are ready to estimate the probability:

```
# treats: true as 1, false as 0
function getAvg(successes::Vec{Bool})::Flt
    return sum(successes) / length(successes)
end

function getProbOfWinningDoorsGame(shouldSwap::Bool=false,
                                   nSimul::Int=10_000)::Flt
    @assert 1e3 <= nSimul <= 1e5 "nSimul must be in range [1e3 - 1e5]"
    return [getResultOfDoorsGame(shouldSwap) for _ in 1:nSimul] |>
getAvg
end

Rnd.seed!(1492)
getProbOfWinningDoorsGame(false), getProbOfWinningDoorsGame(true)
```

```
(0.3354, 0.6668)
```

which ends up to be equivalent to our theoretical calculations.

I suppose you could argue the doing 10,000 computer simulations to estimate the probability is overkill, after all the total number of possibilities cannot be that big for this simple case. Well, I guess you're right. So, let's try again, this time we will list all the possible scenarios and see in how many of them we succeed.

[149]https://docs.julialang.org/en/v1/base/base/#Base.isnothing
[150]https://docs.julialang.org/en/v1/base/base/#?:

```
# list all the possibilities of car location and choice location
function getAll3DoorSets()::Vec{Vec{Door}}
    allDoorSets::Vec{Vec{Door}} = []
    subset::Vec{Door} = Door[]
    for indCar in 1:3, indChosen in 1:3
        subset = [Door(false, false, false) for _ in 1:3]
        subset[indCar].isCar = true
        subset[indChosen].isChosen = true
        push!(allDoorSets, subset)
    end
    return allDoorSets
end
```

Now we change the `return` statements in our `openEligibleDoor!` and `swapChoice`.

```
# open first non-chosen, non-car door
function openEligibleDoor!(doors::Vec{Door})::Vec{Door}
    @assert length(doors) == 3 "need 3 doors"
    indEligible::Int = findfirst(d -> !d.isCar && !d.isChosen, doors)
    doors[indEligible].isOpen = true
    return doors # instead of: return nothing
end

# swap to first non-chosen, non-open door
function swapChoice!(doors::Vec{Door})::Vec{Door}
    @assert length(doors) == 3 "need 3 doors"
    indCurChosen::Int = findfirst(d -> d.isChosen, doors)
    ind2Choose::Int = findfirst(d -> !d.isChosen && !d.isOpen, doors)
    doors[indCurChosen].isChosen = false
    doors[ind2Choose].isChosen = true
    return doors # instead of: return nothing
end
```

Thanks to this small change we can answer our question with this few-liner.

```
map(didTraderWin ∘ openEligibleDoor!, getAll3DoorSets()) |> getAvg,
map(didTraderWin ∘ swapChoice! ∘ openEligibleDoor!,
    getAll3DoorSets()) |> getAvg
```

```
(0.3333333333333333, 0.6666666666666666)
```

> Note, ∘ is a function composition operator[151]that you can obtain by typing `\circ` and pressing Tab. The `map(fn2 ∘ fn1, xs)` means take every `x` of `xs` and send it to `fn1` as an argument. Then send the result of `fn1(x)` as an argument to `fn2`. Finally, present the result of `fn2(fn1(x))` (executed on all `xs`) as a collection.

And that's it. Three methods, three similar results. Time to make that door swap.

> Note, the code presented above may not work for other number of doors scenarios.

[151]https://docs.julialang.org/en/v1/manual/functions/#Function-composition-and-piping

Progress Bar

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[152](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[153].

Problem

While running a time consuming program we may see a progress bar that will provide the user with visual cues as to the course of its execution.

So here is a task for you. Write a computer program that will animate a mock progress bar that goes from 0% to 100% (if you want, make it similar to the one in Figure 6). Don't worry about the complex calculations (the task is just to animate the bar), use sleep[154]before printing the bar in each iteration. In order to redraw a bar in a terminal[155]you may want to read about ANSI escape codes[156]. If the above is too much for you try to use carriage return[157], i.e. `print(progress_bar1)`, `print("\r")` and `print(progress_bar2)`.

[152]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/progress_bar
[153]https://docs.julialang.org/en/v1/stdlib/Pkg/
[154]https://docs.julialang.org/en/v1/base/parallel/#Base.sleep
[155]https://en.wikipedia.org/wiki/Terminal_emulator
[156]https://en.wikipedia.org/wiki/ANSI_escape_code
[157]https://en.wikipedia.org/wiki/Carriage_return

```
Toy program.
It animates a progress bar.
Note: your terminal must support ANSI escape codes.

Continue with the animation? [Y/n]

|||||||||||||||||||||||||||||||||||||||.......... 79%-
```

Figure 6: A mock progress bar (animation works only in an HTML document)

Solution

Let's begin with a function that will return a textual representation of a progress bar.

```
function getProgressBar(perc::Int)::Str
    @assert 0 <= perc <= 100 "perc must be in range [0-100]"
    return "|" ^ perc * "." ^ (100-perc) * "$perc%"
end
```

In order to understand the function we must remember the precedence of mathematical operations [exponentiation (^) before multiplication (*)]. Moreover, we must remember that in the context of strings * is a concatenation[158]operator (it glues two strings into a one longer string), whereas ^ multiplies a string to its left the number of times on its right (i.e. `"a"^3` gives us `"a" * "a" * "a"` so `"aaa"`). `getProgressBar` accepts percentage (`perc`) and draws as many vertical bars as `perc` tells us. The unused spots (`100-perc`) are filed with the placeholders (`"."`). We finish by appending the number itself and the `%` symbol by using string interpolation[159].

Printing a string of 100 characters (actually a bit more) may not look good on some terminals (by default many terminals are 80 characters wide). That is why we may want to limit ourselves to a smaller value

[158]https://docs.julialang.org/en/v1/manual/strings/#man-concatenation
[159]https://docs.julialang.org/en/v1/manual/strings/#string-interpolation

of characters (`maxNumOfChars`) for the progress bar and rescale the percentage (`perc`) accordingly (see below).

```
function getProgressBar(perc::Int)::Str
    @assert 0 <= perc <= 100 "perc must be in range [0-100]"
    maxNumOfChars::Int = 50
    p::Int = round(Int, perc / (100 / maxNumOfChars))
    return "|" ^ p * "." ^ (maxNumChars-p) * "$perc%"
end
```

The above function looses some resolution in translation of `perc` to vertical bars (`|`). However, the percentage is displayed as a number anyway (`"$perc%"`) so it is not such a big problem after all.

Now, we are ready to write the first version of our `animateProgressBar`.

```
function animateProgressBar()::Nothing
    fans::Vec{Str} = ["\\", "-", "/", "-"]
    ind::Int = 1
    for p in 0:100
        println(getProgressBar(p), " ", fans[ind])
        ind = (ind == length(fans)) ? 1 : ind + 1
    end
    println(getProgressBar(100))
    return nothing
end
```

The function is rather simple. For every value of percentage (`for p in 0:100`) we draw the progress bar with a fan that changes into one of four positions (alternating `\`, `-`, `/`, `-` in one place a few times a second will give the impression of a fan). Note, the double backslash[160] character (`"\\"`) in `fans`. The `\` symbol got a particular meaning in programming. It is used to designate that the next character(s) is/are special. For instance `println("and")` will just print the conjunction ‘and’. On the other hand, `println("a\nd")` will print ‘a’ and ‘d’, one below the other, since in Julia `"\n"` stands for newline. To get rid of the special meaning of `"\"` we precede it with another backslash, hence `"\\"`.

[160] https://en.wikipedia.org/wiki/Backslash

OK, let's see what we got.

```
animateProgressBar()
```

```
.................................................. 0% \
.................................................. 1% -
|................................................. 2% /
||................................................ 3% -
||................................................ 4% \
||................................................ 5% -
# part of the output trimmed
||||||||||||||||||||||||||||||||||||||||||||||||.. 96%\
||||||||||||||||||||||||||||||||||||||||||||||||.. 97%-
|||||||||||||||||||||||||||||||||||||||||||||||||. 98%/
|||||||||||||||||||||||||||||||||||||||||||||||||| 99%-
|||||||||||||||||||||||||||||||||||||||||||||||||| 100%\
|||||||||||||||||||||||||||||||||||||||||||||||||| 100%
```

Pretty good. However, there is a small problem. Namely, the output is printed on the screen instantaneously with one line beneath the other. The first problem will be solved with sleep[161]that makes the program wait for a specific number of seconds before executing the next line of code. The second problem will be solved with ANSI escape codes[162]a sequence of characters with a special meaning [as found in the link (see this sentence) to the Wikipedia's page].

```
# the terminal must support ANSI escape codes
# https://en.wikipedia.org/wiki/ANSI_escape_code
function clearPrintout()::Nothing
    #"\033[xxxA" - xxx moves cursor up xxx lines
    print("\033[1A")
    # clears from cursor position till end of display
    print("\033[J")
    return nothing
end

function animateProgressBar()::Nothing
    delayMs::Int = 0
    fans::Vec{Str} = ["\\", "-", "/", "-"]
    ind::Int = 1
    for p in 0:100
```

[161]https://docs.julialang.org/en/v1/base/parallel/#Base.sleep
[162]https://en.wikipedia.org/wiki/ANSI_escape_code

```
        delayMs = rand(100:250)
        println(getProgressBar(p), " ", fans[ind])
        sleep(delayMs / 1000) # sleep accepts delay in seconds
        clearPrintout()
        ind = (ind == length(fans)) ? 1 : ind + 1
    end
    println(getProgressBar(100))
    return nothing
end
```

This time running `animateProgressBar()` will give us the desired result.

As a final touch we will add some functionality for running our script (saved as `progress_bar.jl`) from a terminal[163].

```
function main()::Nothing
    println("Toy program.")
    println("It animates a progress bar.")
    println("Note: your terminal must support ANSI escape codes.\n")

    # y(es) - default choice (also with Enter), anything else: no
    println("Continue with the animation? [Y/n]")
    choice::Str = readline()
    if lowercase(strip(choice)) in ["y", "yes", ""]
        animateProgressBar()
    end

    println("\nThat's all. Goodbye!")

    return nothing
end

if abspath(PROGRAM_FILE) == @__FILE__
    main()
end
```

Although not strictly required in Julia, the `main` function is by convention a starting point of any (big or small) program (in many programming languages). Whereas the `if abspath` part makes sure that our `main` function is run only if the program was called directly from the terminal, i.e.

[163]https://en.wikipedia.org/wiki/Terminal_emulator

```
julia progress_bar.jl
```

will run it, while:

```
julia other_file_that_imports_progress_bar.jl
```

will not.

Pascal's Triangle

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[164](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[165].

Problem

Imagine that you are a basketball coach in a local school. A basketball team is composed of 5 players. Nine kids attends your classes. You would like the kids to play in every possible configuration so that you could choose the best team to represent the school next month. You wonder how many different teams of 5 players can you compose out of 9 candidates.

Such a question could be answered with the binomial[166]function or Pascal's triangle[167].

So here is a task for you, write a program that will display the Pascal's triangle that will help you to answer the question mentioned above.

Compare its output with `binomial`.

The function can display a bare (right) triangle

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
```

---

[164] https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/pascals_triangle

[165] https://docs.julialang.org/en/v1/stdlib/Pkg/

[166] https://docs.julialang.org/en/v1/base/math/#Base.binomial

[167] https://en.wikipedia.org/wiki/Pascal%27s_triangle

or if you like challenges a slightly formatted output (it doesn't have to be exact):

```
     1
    1 1
   1 2 1
  1 3 3 1
      Δ
```

The above indicates how many pairs of 2 people out of 3 candidates can we get (e.g. in order to play doubles in tennis).

Solution

If you read the Wikipedia's description of the Pascal's triangle[168]then you may have noticed this cool animation (see below).

[168]https://en.wikipedia.org/wiki/Pascal%27s_triangle

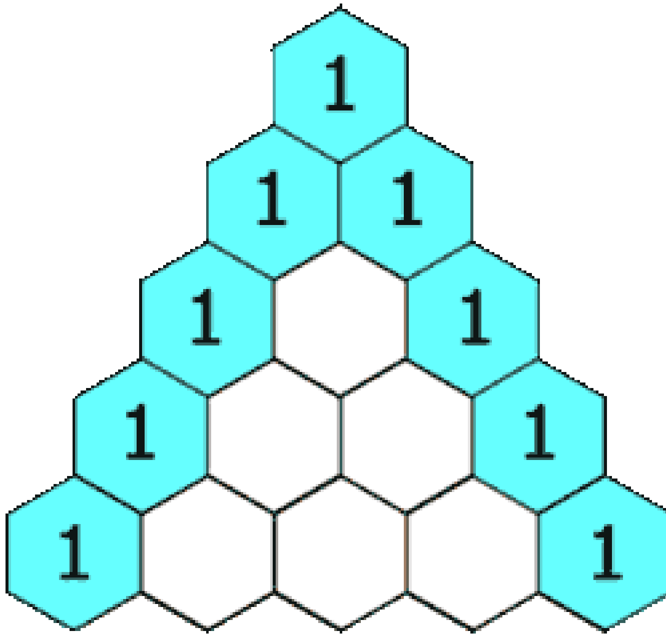


Figure 7: Construction of a Pascal's triangle. Source: Wikipedia[169] (public domain, used in accordance with the Licensing section, animation works only in an HTML document).

It indicates that in order to create a new row of the triangle you just take the previous row and add pair of its elements together to create the next row. Finally, at the edges of the new row you place 1s. So let's start by doing just that.

```
function getSumOfPairs(prevRow::Vec{Int})::Vec{Int}
    @assert all(prevRow .> 0) "every element of prevRow must be > 0"
    return [a + b for (a, b) in zip(prevRow, prevRow[2:end])]
end

function getRow(prevRow::Vec{Int})::Vec{Int}
    @assert length(prevRow) > 1 "length(prevRow) must be > 1"
    sumsOfPairs::Vec{Int} = getSumOfPairs(prevRow)
    return [1, sumsOfPairs..., 1]
end
```

[169] https://en.wikipedia.org/wiki/File:PascalTriangleAnimated2.gif

The first function (`getSumOfPairs`) creates pairs of values based on the previous row (`prevRow`). It returns a vector of tuples of `(a, b)` that are the elements from the zipped vectors, e.g.

```
zip([1, 2, 3, 4], [2, 3, 4]) |> collect
```

```
(1, 2)
```

```
(2, 3)
```

```
(3, 4)
```

Notice, that the second argument to `zip` is its first element (`[1, 2, 3, 4]`) with shift 1 (just like `prevRow[2:end]` in the body of `getSumOfPairs`) and the parallel elements from both the vectors are `zip`ped together as long as there are pairs. Next, the numbers in a pair are added (`a + b` in the body of `getSumOfPairs`). Finally, `getRow` uses the `sumsOfPairs` and adds 1s on the edges. The `...` in `getRow` unpacks elements from a vector, i.e. `[1, [3, 2]..., 1]` becomes `[1, 3, 2, 1]`. All in all, we got a pretty faithful translation of the algorithm (from Figure 7) to Julia.

Now we are ready to build the triangle row by row.

```
function getPascalTriangle(n::Int)::Vec{Vec{Int}}
    @assert 0 <= n <= 10 "n must be in range [0-10]"
    triangle::Dict{Int, Vec{Int}} = Dict(0 => [1], 1 => [1, 1])
    if !haskey(triangle, n)
        for row in 2:n
            triangle[row] = getRow(triangle[row-1])
        end
    end
    return [triangle[k] for k in 0:n]
end
```

We define our triangle with initial two rows. Next, we move downwards through the possible triangle rows (`for row in 2:n`) and build the next row based on the previous one `getRow(triangle[row-1])`. All that's left to do is to `return` the

triangle as a vector of vectors (`Vec{Vec{Int}}`) which will give us a right triangle printed in the output by default. For instance, let's get the Pascal's triangle from Figure 7 .

```
getPascalTriangle(4)
```

```
[1]
```

```
[1, 1]
```

```
[1, 2, 1]
```

```
[1, 3, 3, 1]
```

```
[1, 4, 6, 4, 1]
```

Pretty neat, we could stop here or try to add some text formatting. In order to do that we will need some way to determine the length of an integer when printed (`getNumLen` below) as well as the maximum number length in a Pascal's triangle. The latter can be done by examining its last (longest) row, hence `getMaxNumLen` below accepts a vector.

```
function getNumLen(n::Int)::Int
    return n |> string |> length
end

function getMaxNumLen(v::Vec{Int})::Int
    return map(getNumLen, v) |> maximum
end
```

Once we got it, we need a way to center a number or a row (represented as a string).

```
function center(sth::A, endLength::Int)::Str where A<:Union{Int, Str}
    s::Str = string(sth)
    len::Int = length(s)
```

```
    @assert endLength > 0 && len > 0 "both endLength and len must be >
0"
    @assert endLength >= len "endLength must be >= len"
    diff::Int = endLength - len
    leftSpaceLen::Int = div(diff, 2) # divide by 2, round down
    rightSpaceLen::Int = diff - leftSpaceLen
    return " " ^ leftSpaceLen * s * " " ^ rightSpaceLen
end
```

In order to center its input (`sth` - an integer or a string, as stated in `A<:Union{Int, Str}`) the function determines the difference (`diff`) between the length of the desired result (`endLength`) and the actual length of `s` (`len`). The difference is split roughly in half (`leftSpaceLen` and `rightSpaceLen`) and glued together with `s` using string concatenation (*) and exponentiation (^) that we met in Section 17.2 . Due to the limited resolution offered by the text display of a terminal the result is expected to be slightly off on printout, but I think we can live with that.

Time to format a row.

```
function getFmtRow(
    row::Vec{A}, numLen::Int, rowLen::Int)::Str where A<:Union{Int, Str}
    fmt(num) = center(num, numLen)
    formattedRow::Str = join(map(fmt, row), " ")
    return center(formattedRow, rowLen)
end
```

For that we just `map` a formatter (`fmt`) over every element of the `row` and `join` the resultant vector of strings intercalating its elements with space (`" "`). Finally, we center the `formattedRow` to the desired length (`rowLen`).

All that's left to do is to get a formatted triangle.

```
function getFmtPascTriangle(n::Int, k::Int)::Str
    @assert n >= 0 && k >= 0 "n and k must be >= 0"
    @assert n <= 10 && k <= 10 "n and k must be <= 10"
    @assert n >= k "n must be >= k"
    triangle::Vec{Vec{Int}} = getPascalTriangle(n)
    lastRow::Vec{Int} = triangle[end]
    maxNumWidth::Int = getMaxNumLen(lastRow) + 1
```

```
    lastRowWidth::Int = (n+1) * maxNumWidth + n
    fmtRow(row) = getFmtRow(row, maxNumWidth, lastRowWidth)
    formattedTriangle::Str = join(map(fmtRow, triangle), "\n")
    indicators::Vec{Str} = fill(" ", n+1)
    indicators[k+1] = "Δ"
    return formattedTriangle * "\n" * fmtRow(indicators)
end
```

Since `getFmtPascTriangle` is a graphical equivalent of `binomial(n, k)` then it's only fitting to accept the two letters (`n` and `k`) as its input. We begin by obtaining the `triangle`, its `lastRow` and based on it the maximum width of a number in the triangle (`maxNumWidth`, the magic number `+1` is to produce more visually pleasing output). Next, we determine the width of the last row. Notice the `n+1` part (here and below in the function) as well as `k+1` part later on. In, general Julia and humans count elements starting from 1, whereas a Pascal's triangle is 0 indexed, hence we added `+1` to help us translate one system into the other. Anyway, the length of `lastRow` (the longest row in `triangle`) is the number of digits in the row (`(n+1)`) times the width of a formatted digit (`maxNumWidth`) plus the number of spaces between the formatted digits. The number of spaces is 1 less than the number of slots, e.g., humans got 5 fingers, and 4 spaces between them, hence here we used `+n` since the number of digits was `(n+1)`. Next, we obtain the `formattedTriangle` by `map`ing `fmtRow` on its each row and separating the rows with newlines (`\n`). We finish, by adding the indicator (`"Δ"`) under our `k` and voila, we are finally ready to answer our question.

```
# how many different teams of 5 players
# can we compose out of 9 candidates?
getFmtPascTriangle(9, 5)
```

```
                  1
                1    1
              1    2    1
           1    3    3    1
        1    4    6    4    1
      1    5   10   10    5    1
    1    6   15   20   15    6    1
 1    7   21   35   35   21    7    1
1    8   28   56   70   56   28    8    1
```

```
  1    9   36   84  126  126   84   36    9    1
                          Δ
```

Wow, 126. Who would have thought. Just in case let's compare the output with the built in `binomial` (compare with the last row of the triangle).

```
binomial.(9, 0:9)
```

```
[1, 9, 36, 84, 126, 126, 84, 36, 9, 1]
```

So, I guess our triangle works right. Actually the two edge cases are easy enough for us to check them in our heads. For instance, how many different teams of 0 players can we compose out of 9 candidates? Well, there is only one way to do that (`1` in the bottom row, first from the left), by not choosing any player at all. And how many different teams of 1 player can we compose of 9 candidates? Well, nine teams (`9` in the bottom row, second from the left) because each player would compose one separate team.

Lattice Paths

In this chapter I used the following libraries.

```
import CairoMakie as Cmk # external library
```

Still, once you read the problem description you may decide to do otherwise.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[170](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[171].

Problem

This exercise was inspired by the following Euler Project problem[172], but it was modified by me.

Take a look at Figure 8 below. You will find a big square build of 2x2 smaller squares. Your task is to start at the top left corner of the big square and go to the bottom right corner of it using only right and down arrows that span a small square side length.

[170]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/lattice_paths

[171]https://docs.julialang.org/en/v1/stdlib/Pkg/

[172]https://projecteuler.net/problem=15

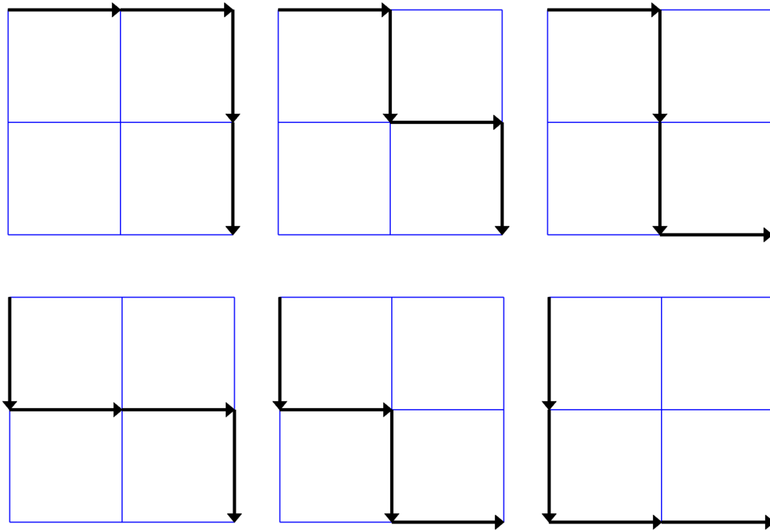


Figure 8: Lattice paths on a 2x2 grid.

For the said problem there are exactly 6 paths (as depicted in Figure 8). But think about a big square composed of 3x3 little squares.

How many different paths are there? Can you draw these paths? Well, use Julia to find out.

Solution

The lattice paths[173]is actually a problem from the field of combinatorics[174]. In our particular case (3x3 grid) it could be solved (number of paths calculation) with a Pascal's triangle (covered in Section 18) or the build-in `binomial` function (`binomial(nRows+nCols, nRows)`). Since I'm not a mathematician, then for a detailed explanation I refer you to this Wikipedia's entry[175].

[173]https://projecteuler.net/problem=15
[174]https://en.wikipedia.org/wiki/Combinatorics
[175]https://en.wikipedia.org/wiki/Lattice_path#Combinations_and_NE_lattice_paths

Still, for practical reasons I like to use computer calculations to help me with my understanding. Let's start small. First, analyze the pictures in Figure 9 and Figure 10 .

> ***Note:*** The solution presented here is practical only for small girds (up to 4x4 lattice, 70 paths to calculate and to draw). The original problem[176](20x20 grid) got `binomial(40, 20)` = 137846528820 possible routes between the top-left and bottom-right corner. That's far too many to calculate with the presented method and to draw in one figure.

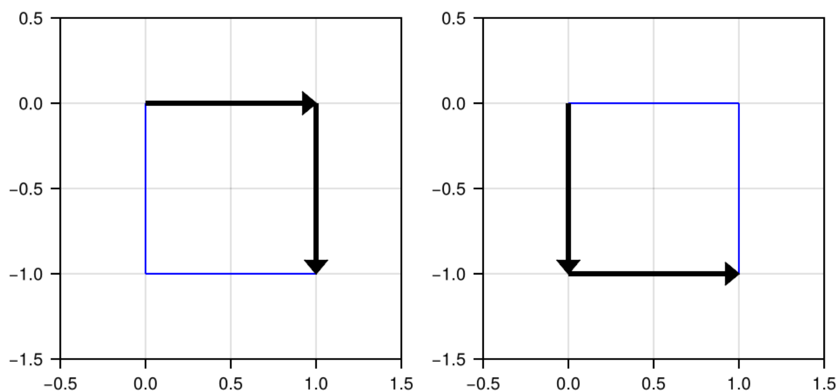


Figure 9: Lattice paths on a 1x1 grid in Cartesian coordinate system. Blue lines designate individual small squares.

[176]https://projecteuler.net/problem=15

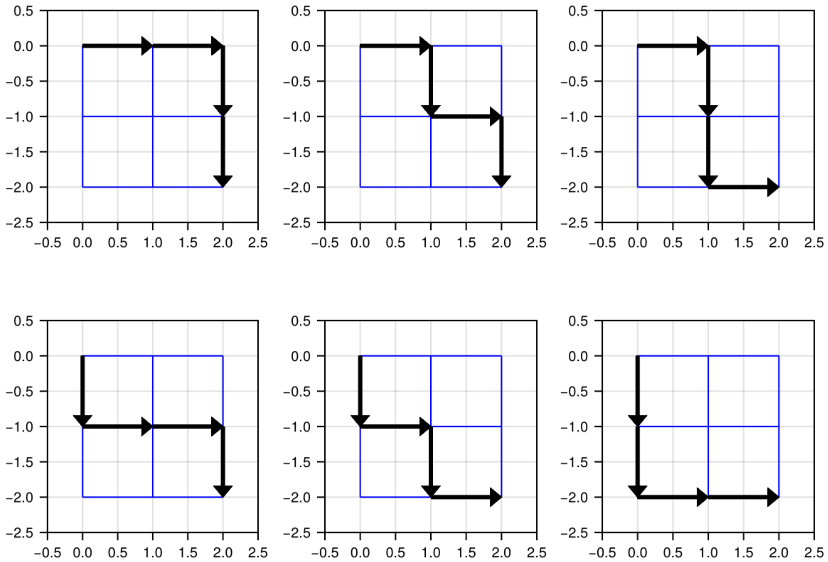


Figure 10: Lattice paths on a 2x2 grid in Cartesian coordinate system. Blue lines designate individual small squares

A few points of notice (make sure they agree on both Figure 9 and Figure 10):

1) the top left corner could be considered to be the center of our Cartesian coordinate system[177] with the location (0, 0);
2) the bottom right corner could be located within that system at the position (nRows, -nRows) or (nCols, -nCols) of our grid (each small square in Figure 9 and Figure 10 got the side length = 1 and builds rows and columns of a large square);
3) The number of arrows used to reach the destination (the bottom right corner) is always the same for each path and it is equal to nRows+nCols (or nRows*2 or nCols*2);

Equipped with this knowledge, we can finally do something useful.

[177] https://en.wikipedia.org/wiki/Cartesian_coordinate_system

```
const Pos = Tuple{Int, Int}
const Mov = Tuple{Int, Int}

const RIGHT = (1, 0)
const DOWN = (0, -1)
const MOVES = [RIGHT, DOWN]
const STARTPOINT = (0, 0)

function add(position::Pos, move::Mov)::Pos
    return position .+ move
end

function add(positions::Vec{Pos}, moves::Vec{Mov}=MOVES)::Vec{Pos}
    @assert !isempty(positions) "positions cannot be empty"
    @assert !isempty(moves) "moves cannot be empty"
    result::Vec{Pos} = []
    for p in positions, m in moves
        push!(result, add(p, m))
    end
    return result
end
```

We begin by defining a few constants. The type synonyms: `Pos` (shortcut for position), `Mov` (shortcut for move) will save us some typing later on. Whereas, `RIGHT` (shift by 1 unit along X-axis) and `DOWN` (shift by 1 unit along Y-axis) are the arrow coordinates in the Cartesian coordinate system (it starts at the top left corner, `(0, 0)`, of a big square), which together form a vector of available `MOVES`.

Next, we define the `add` function. Its first version, aka method, allows to add a position (like our `STARTPOINT`) to a move (like the move to the `RIGHT`). In some programming languages we would have to type something like `return (position[1] + move[1], position[2] + move[2])` but in Julia we just use a broadcasting operator `.+` to add the tuples element-wise. The second version of `add` takes a vector of any starting `positions` and any possible `moves` and adds each move to each starting position (using simplified nested for loop syntax we met in Section 2.2) to get the new positions (places on the grid) after the moves.

Let's put the above functions to good use.

```
function getFinalPositions(nRows::Int)::Vec{Pos}
    @assert 0 < nRows < 5 "nRows must be in the range [1-4]"
    finalPositions::Vec{Pos} = [STARTPOINT] # top left corner
    for _ in 1:(nRows*2) # *2 - because of columns
        finalPositions = add(finalPositions, MOVES)
    end
    return finalPositions
end

function getNumOfPaths(nRows::Int)::Int
    @assert 0 < nRows < 5 "nRows must be in the range [1-4]"
    target::Pos = (nRows, -nRows) # bottom right corner
    positions::Vec{Pos} = getFinalPositions(nRows)
    return filter(pos -> pos == target, positions) |> length
end

getNumOfPaths(3) # the same result as: binomial(6, 3)
```

```
20
```

`getFinalPositions` accepts `nRows` which is the number of small squares in a column of, e.g. Figure 10 . Next, it starts at the top left corner (`finalPositions::Vec{Pos} = [STARTPOINT]`) and `moves` away from it. The number of moves is equal to `nRows*2` and we always go in each of both the directions (`RIGHT` and `DOWN` which are in `moves`) thanks to the previously defined `add` function. Finally, we `return` the `finalPositions` that we get after we made all the possible moves. Next, in `getNumOfPaths`, we choose only those `positions` that land us in the bottom right corner (`target`) by using `filter`. The `length` of our vector is the number of paths we were looking for.

OK, now let's think how to draw it. For once, we could reuse the already written functions (`add` and `getFinalPositions`) like so (we will modify the functions a little bit):

```
function makeOneStep(prevPaths::Vec{Path}, moves::Vec{Mov}
=MOVES)::Vec{Path}
    @assert !isempty(prevPaths) "prevPaths cannot be empty"
    @assert !isempty(moves) "moves cannot be empty"
    result::Vec{Path} = []
    for path in prevPaths, move in moves
        push!(result, [path..., add(path[end], move)])
```

```
    end
    return result
end

function getPaths(nRows::Int)::Vec{Path}
    @assert 0 < nRows < 5 "nRows must be in the range [1-4]"
    target::Pos = (nRows, -nRows) # bottom right corner
    result::Vec{Path} = [[STARTPOINT]] # top left corner
    for _ in 1:(nRows*2) # *2 - because of columns
        result = makeOneStep(result, MOVES)
    end
    return filter(path -> path[end] == target, result)
end
```

We begin with `makeOneStep` (analogue to `add(positions, moves)`), a function that makes every possible step (`moves`) from the last known location of every path in `prevPaths`. BTW, notice how defining the type synonyms paid off. Without them `Vec{Mov}` would be `Vector{Tuple{Int, Int}}` (it isn't all that bad), but `Vec{Path}` would grow to `Vector{Vector{Tuple{Int, Int}}}`, which is a little monster. Anyway, thanks to the double `for` loop we `add` every possible `move` (here `RIGHT` or `DOWN`) to the last know location of a `path` (`path[end]`, which is a `Pos`) and append it (`push!`) to the `result` (`path...` copies the previous vector, so `[path..., add(path[end], move)]` yields the previous `path` with a position after that `move`, e.g. `[(0, 0)]` becomes `[(0, 0), (1, 0)]`).

Once we know how to `makeOneStep` (remember we test both directions/`moves` at once, so we branch our paths into two separate paths at each step we take) time to take `nRows*2` steps (with `for`) so that we can see which ones will eventually create the paths that will lead us to our final position (`path[end] == target`). As you have guessed this is exactly what `getPaths` does.

Let's see how we did so far (simple minimal test, locate the points depicted with tuples on Figure 9).

```
getPaths(1)
```

```
[(0, 0), (1, 0), (1, -1)]
[(0, 0), (0, -1), (1, -1)]
```

I don't know about you, but I'm pretty satisfied with the result.

OK, once we got the path, i.e. the stop points between which we draw the lines, making a Figure that depicts them should be a breeze. This could be done with, e.g. CairoMakie's arrows2d[178]function. However, there's a small problem here, one look at the documentation and we see that the function requires two arguments, namely `points` (start points) and `directions` (like `RIGHT` and `DOWN`) and not a `path` which is starting point, pit stop, pit stop, […], end point. No biggie, we can get the directions in no time.

```
function getDirection(p1::Pos, p2::Pos)::Mov
    return p2 .- p1
end

function getDirections(path::Path)::Vec{Mov}
    directions::Vec{Mov} = []
    for i in eachindex(path)[2:end]
        push!(directions, getDirection(path[i-1], path[i]))
    end
    return directions
end
```

To `getDirection` we just subtract the position of a previous point on the line (`p1`) from the position of a next point on it (`p2`). By extension, in order to `getDirections` we do that for all the points. The last function's body (`getDirections`) could be also replaced with the following one liner: `return map(getDirection, path[1:end-1], path[2:end])` (similar to what we did in Section 2.2). Feel free to choose the version whose meaning is clearer to you.

Finally, time to draw.

```
import CairoMakie as Cmk

function addGrid!(ax::Cmk.Axis,
```

[178]https://docs.makie.org/stable/reference/plots/arrows#arrows

```
                 xmin::Int=0, xmax::Int=2,
                 ymin::Int=-2, ymax::Int=0)::Nothing
    @assert xmin < xmax "xmin must be < xmax"
    @assert ymin < ymax "ymin must be < ymax"
    for yCut in ymin:ymax
        Cmk.lines!(ax, [xmin, xmax], [yCut, yCut], color=:blue,
linewidth=1)
    end
    for xCut in xmin:xmax
        Cmk.lines!(ax, [xCut, xCut], [ymin, ymax], color=:blue,
linewidth=1)
    end
    return nothing
end

function drawPaths(paths::Vec{Path}, nCols::Int)::Cmk.Figure
    @assert length(paths) % nCols == 0 "length(paths) % nCols is not 0"
    r::Int, c::Int = 1, 1 # r - row, c - column of subFig on Figure
    sp::Flt = 0.5 # extra space on X/Y axis for better look
    xmin::Int, xmax::Int = paths[1][1][1], paths[1][end][1]
    ymax::Int, ymin::Int = paths[1][1][2], paths[1][end][2]
    fig::Cmk.Figure = Cmk.Figure()
    for path in paths
        ax = Cmk.Axis(fig[r, c],
                      limits=(xmin-sp, xmax+sp, ymin-sp, ymax+sp),
                      aspect=1, xticklabelsize=10, yticklabelsize=10)
        Cmk.hidespines!(ax)
        Cmk.hidedecorations!(ax)
        directions::Vec{Mov} = getDirections(path)
        points::Vec{Pos} = path[1:end-1]
        addGrid!(ax, xmin, xmax, ymin, ymax)
        Cmk.arrows2d!(ax, points, directions)
        if c == nCols
            r += 1
            c = 1
        else
            c += 1
        end
    end
    Cmk.rowgap!(fig.layout, Cmk.Fixed(1))
    Cmk.colgap!(fig.layout, Cmk.Fixed(1))
    return fig
end
```

We begin with a helper function `addGrid!` that does what it promises (draws blue lines in sub-figures of our main figure, see Figure 10). Then we move to `drawPaths` that draws each `path` of `paths` (`for path in paths`) on a separate sub-figure (its location is specified by `fig[r,`

c]). A `path` is depicted with a set of arrows (`arrows2d!`) on the blue grid (`addGrid!`). We hid the Cartesian coordinate system (`Cmk.hidespines!` and `Cmk.hidedecorations!`) because in the end we don't care about it that much. Moreover, we narrowed the empty space between the sub-figures (`Cmk.rowgap!` and `Cmk.colgap!`). The rest of the code in this snippet is just the necessary, housekeeping that helps us achieve our goal, which is depicted below.

```
paths = getPaths(3)
drawPaths(paths, 5)
```

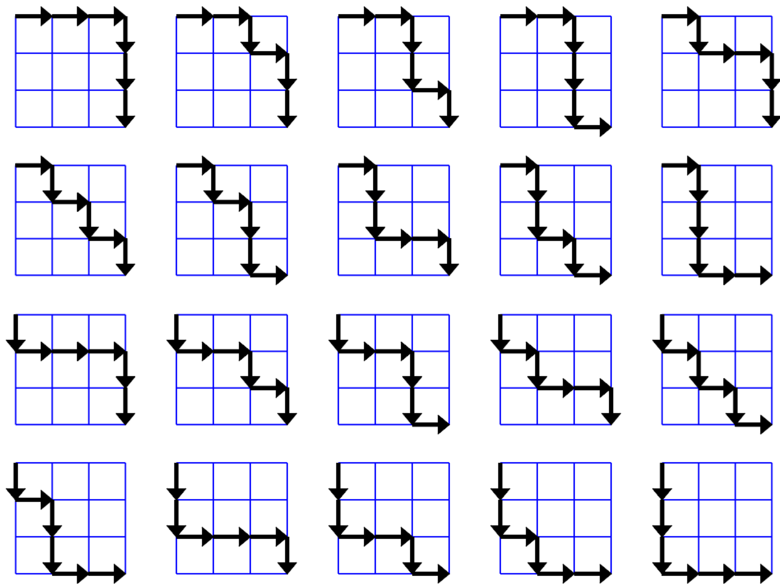


Figure 11: Lattice paths on a 3x3 grid.

All is revealed. Behold the twenty paths on a 3x3 grid.

Stem and Leaf Plot

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[179](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[180].

Problem

In statistics a useful technique used to visualize the distribution of data is a histogram[181]. A bit less popular, although easier to implement with text display is stem and leaf plot[182].

So here is a task for you:

- read the Wikipedia's description of how the plot is constructed
- write the function that displays stem and leaf plot for positive integers
- extend it to work also with negative integers
- extend it to work also with floats

As a minimal test, make sure it works correctly on the examples from the Wikipedia's web page[183], i.e.

```
# prime numbers below 100
stemLeafTest1 = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37,
                41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
# example from the Construction section
stemLeafTest2 = [44, 46, 47, 49, 63, 64, 66, 68, 68, 72, 72, 75,
```

[179]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/stem_and_leaf

[180]https://docs.julialang.org/en/v1/stdlib/Pkg/

[181]https://en.wikipedia.org/wiki/Histogram

[182]https://en.wikipedia.org/wiki/Stem-and-leaf_display

[183]https://en.wikipedia.org/wiki/Stem-and-leaf_display

```
                76, 81, 84, 88, 106]
# another example from the Construction section
stemLeafTest3 = [-23.678758, -12.45, -3.4, 4.43, 5.5, 5.678,
                16.87, 24.7, 56.8]
```

Notice that if you broaden the range of input numbers (e.g. by appending 300 or 400, to `stemLeafTest1`) you will still get a printout, but the plot won't be so pretty.

Solution

Let's start with a helper function that returns the number of characters that compose a number.

```
function howManyChars(num::Int)::Int
    return num |> string |> length
end
```

The function is quite simple, it sends (`|>`) a number (`num`) to `string` (converts a number to its textual representation) and redirects the result (`|>`) to `length`. I find this form clearer that the equivalent `length(string(num))` or `string(num) |> length` or `(length ∘ string)(num)` (`∘` is a function composition operator[184]that you obtain by typing `\circ` and pressing Tab).

Time for a test run.

```
map(howManyChars, [5, -8, -11, 303])
```

```
[1, 2, 3, 3]
```

Appears to be working fine.

Now let's write a function that takes a vector of integers and returns the number of characters in the longest of them (we will need it to determine the width of the stem later on).

[184]https://docs.julialang.org/en/v1/manual/functions/#Function-composition-and-piping

```
function getMaxLengthOfNum(nums::Vec{Int})::Int
    maxLen::Int = map(howManyChars, nums) |> maximum
    return max(2, maxLen)
end
```

> Note. Instead of `map(howManyChars, nums)` above we could have just used `map(length ∘ string, nums)`. This would save us some typing (no need to define `howManyChars` in the first place), but made the code a bit more cryptic at first read.

Again, a piece of cake, we just use `map` to apply `howManyChars` to every number in a vector (`nums`) and get the length of the longest number by sending (`|>`) the lengths to `maximum`. Notice, that the function doesn't return the expected `maxLen`. This is because in a moment, we will write `getStemAndLeaf(num::Int, maxLenOfNum::Int)` that brakes a number into two parts: stem and leaf. It will require `maxLenOfNum` to be at least 2 (so that at least one digit serves as a stem and one as a leaf), hence we `return max(2, maxLen)`.

```
function getStemAndLeaf(num::Int, maxLenOfNum::Int)::Tuple{Str, Str}
    @assert maxLenOfNum > 1 "maxLenOfNum must be greater than 1"
    @assert howManyChars(num) <= maxLenOfNum
        "character count in num must be <= maxLenOfNum"
    numStr::Str = lpad(abs(num), maxLenOfNum, "0")
    stem::Str = numStr[1:end-1] |> string
    leaf::Str = numStr[end] |> string
    stem = parse(Int, stem) |> string #1
    stem = num < 0 ? "-" * stem : stem #2
    stem = lpad(stem, maxLenOfNum-1, " ") #3
    return (stem, leaf)
end
```

We begin with `lpad`. This function converts its first input (`abs(num)`) to string of a given length (`maxLenOfNum`). It adds a padding (`"0"`) to the left side of the result (if necessary) in order to obtain the string with a desired number of characters. Next, we proceed to obtain the `stem` which contains all the characters from `numStr`, except the last one (`end-1`). The `|> string` makes sure that the end result is `Str` (since, e.g. `stem` from `"21"` would be `'2'` which is of type `Char`).

Similarly, we produce `leaf` by taking the last character of `numStr`. We could stop here, and it would likely work fine for a positive integer. However, handling broader range of inputs (`num` and `maxLenOfNum`) requires some further `stem` processing. Hence the lines designated with `#1`-`#3` that were added in later iterations of `getStemAndLeaf`.`#1` removes superfluous 0s from the left side of the string (e.g. `"001"` becomes `"1"` and `"00"` becomes `"0"`). `#2` adds `"-"` sign if the input (`num`) was negative. `#3` aligns the text (`stem`) to the right. It does so by adding spaces (`" "`) to the left site of `stem`. All that's left to do is to `return` our `stem` and `leaf` and see how it works for some exemplary inputs.

```
Dict(n => getStemAndLeaf(n, 3) for n in [-12, -3, 3, 8, 10, 145])
```

```
Dict{Int64, Tuple{String, String}} with 6 entries:
  -12 => ("-1", "2")
  10 => (" 1", "0")
  145 => ("14", "5")
  -3 => ("-0", "3")
  8 => (" 0", "8")
  3 => (" 0", "3")
```

Time to write `getLeafCounts` a function that for a vector of numbers returns a mapping (`Dict`) between stems (keys) and leaves (values).

```
# returns Dict{stem, [leaves]}
function getLeafCounts(nums::Vec{Int},
    maxLenOfNum::Int)::Dict{Str, Vec{Str}}
    @assert length(unique(nums)) > 1 "numbers musn't be the same"
    counts::Dict{Str, Vec{Str}} = Dict()
    for num in nums
        stem, leaf = getStemAndLeaf(num, maxLenOfNum) # for's local vars
        if haskey(counts, stem)
            counts[stem] = push!(counts[stem], leaf)
        else
            counts[stem] = [leaf]
        end
    end
    return counts
end
```

First, we initialize an empty `Dict` (`counts`) that will hold our result. Next, we brake each number (`for num in nums`) into `stem` and `leaf` parts. If the `counts` already contains such a `stem` (`haskey(counts, stem)`), then we add the `leaf` to the vector of already existing leaves (`push!(counts[stem], leaf)`). Otherwise (`else`), we add a `leaf` as a 1-element vector (`[leaf]`) for a given `stem`. Finally, we return the `counts`.

Let's see how it works.

```
# prime numbers below 100
primesLeafCounts = getLeafCounts(
    [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
        53, 59, 61, 67, 71, 73, 79, 83, 89, 97],
    2
)
```

```
Dict{String, Vector{String}} with 10 entries:
  "8" => ["3", "9"]
  "4" => ["1", "3", "7"]
  "1" => ["1", "3", "7", "9"]
  "5" => ["3", "9"]
  "0" => ["2", "3", "5", "7"]
  "2" => ["3", "9"]
  "6" => ["1", "7"]
  "7" => ["1", "3", "9"]
  "9" => ["7"]
  "3" => ["1", "7"]
```

Looks, alright. Time to pretty print the result. First, let's get a formatted row.

```
function getStemLeafRow(key::Str, leafCounts::Dict{Str, Vec{Str}})::Str
    row::Str = key * "|"
    if haskey(leafCounts, key)
        row *= sort(leafCounts[key]) |> join
    end
    return row * "\n"
end
```

We define our `row` as a string that contains the `key` and separator. If our `leafCounts` contains a given `key` then we append its sorted values

concatenated together with `join` function (e.g., `["1", "1", "3"] |> join` becomes `"113"`). We return `row` with a newline character (`\n`).

Time for the whole stem and leaf plot.

```
function getStemLeafPlot(nums::Vec{Int})::Str
    maxLenOfNum::Int = getMaxLengthOfNum(nums)
    leafCounts::Dict{Str, Vec{Str}} = getLeafCounts(nums, maxLenOfNum)
    low::Int, high::Int = extrema(nums)
    testedStems::Dict{Str, Bool} = Dict()
    result::Str = ""
    for num in low:1:high
        stem, _ = getStemAndLeaf(num, maxLenOfNum)
        if haskey(testedStems, stem)
            continue
        end
        result *= getStemLeafRow(stem, leafCounts)
        testedStems[stem] = true
    end
    return result
end
```

At the onset, we define a few variables. Some of them deserve a short explanation. `low` and `high` are the two `extrema` (minimum and maximum) of our numbers (`nums`). `testedStems` will contain the keys from `leafCounts`, i.e. the stems from our stem-leaf plot that rows has been already obtained. Next, we use `for` loop to travel through all the numbers in our range (`low` to `high`). For each tested number (`num`) we get its `stem`. If the `stem` was already obtained (`if haskey`) we `continue` to another `for` loop iteration. Otherwise, we add the row to our `result` (`result *= getStemLeafRow`) and insert the `stem` among the already visited (`testedStems[stem] = true`). When we finish we return the whole stem-leaf-plot (`return result`).

And that's it. Let's see how it works on Wikipedia's examples[185]. First, prime numbers below 100:

```
getStemLeafPlot(stemLeafTest1)
```

[185] https://en.wikipedia.org/wiki/Stem-and-leaf_display

```
0|2357
1|1379
2|39
3|17
4|137
5|39
6|17
7|139
8|39
9|7
```

Now, the numbers from the Construction section:

```
getStemLeafPlot(stemLeafTest2)
```

```
 4|4679
 5|
 6|34688
 7|2256
 8|148
 9|
10|6
```

All that's left to do is to adjust our function for the example with floats.

```
function getStemLeafPlot(nums::Vec{Flt})::Str
    ints::Vec{Int} = round.(Int, nums)
    return getStemLeafPlot(ints)
end
```

And voila:

```
getStemLeafPlot(stemLeafTest3)
```

```
-2|4
-1|2
-0|3
 0|466
 1|7
 2|5
 3|
```

```
4|
5|7
```

It appears to be working as intended so I think we can finish here.

Canvas

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[186](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[187].

Problem

Create a simple pixel-art terminal[188]graphics. You may, e.g. draw a house on a meadow that is similar to the one below.

[186]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/canvas

[187]https://docs.julialang.org/en/v1/stdlib/Pkg/

[188]https://en.wikipedia.org/wiki/Terminal_emulator

Figure 12: An exemplary pixel-art terminal graphics made with Julia.

Solution

The first question to answer is how to represent our `canvas`. Here, we'll go with a matrix of strings which we will tint by placing space characters (`"  "`) on a background colored with ANSI escape codes[189].

```
const PIXEL = "  "

# "\x1b[48:5:XXXm" sets background color to XXX color code (256-color
mode)
const BG_COLORS = Dict(
    :gray => "\x1b[48:5:8m",
    :white => "\x1b[48:5:15m",
```

[189] https://en.wikipedia.org/wiki/ANSI_escape_code#Colors

```
    :red => "\x1b[48:5:160m",
    :yellow => "\x1b[48:5:11m",
    :blue => "\x1b[48:5:12m",
    :darkblue => "\x1b[48:5:20m",
    :green => "\x1b[48:5:35m",
    :black => "\x1b[48:5:0m",
    :brown => "\x1b[48:5:88m",
)

# "\x1b[0m" resets background color to default value
# default color: "\x1b[48:5:13m" - pink
function getBgColor(color::Symbol,
                    colors::Dict{Symbol, Str}=BG_COLORS,
                    defColor::Str="\x1b[48:5:13m")::Str
    return get(colors, color, defColor) * PIXEL * "\x1b[0m"
end

canvas = fill(getBgColor(:gray), 30, 60)
```

> Note. Using `const` with mutable containers like vectors or dictionaries allows to change their contents later on, e.g., with `push!`. So the `const` used here is more like a convention, a signal that we do not plan to change the containers in the future. If we really wanted an immutable container then we should consider a(n) (immutable) tuple. Anyway, some programming languages suggest that `const` names should be declared using all uppercase characters to make them stand out. Here, I follow this convention.

Next, we want a way to properly display canvas (`printCanvas`) and to clear it (`clearCanvas!`). This last method will allow us to erase an incorrect drawing and try again and again if we need to.

```
function printCanvas(cvs::Matrix{Str}=canvas)::Nothing
    nRows, _ = size(cvs)
    for r in 1:nRows
        println(cvs[r, :] |> join)
    end
    return nothing
end

function clearCanvas!(cvs::Matrix{Str}=canvas)::Nothing
    cvs .= getBgColor(:gray)
```

```
    return nothing
end
```

Now, to draw a picture we will need a few shapes, most likely: a rectangle, a triangle and an oval/circle. A shape will be represented as a vector of positions in our matrix (`canvas`). The positions need to be dyed with a specific color to visualize an object. Let's start with a rectangle as this should be the easiest shape to obtain.

```
const COORD_ORIGIN = (1, 1) # origin or the coordinate system (row, col)

const Pos = Tuple{Int, Int} # position, (row, col) in canvas

function getRectangle(width::Int, height::Int)::Vec{Pos}
    @assert width >= 2 "width must be >= 2"
    @assert height >= 2 "height must be >= 2"
    rectangle::Vec{Pos} = Vec{Pos}(undef, width * height)
    rowStart::Int, colStart::Int = COORD_ORIGIN
    i::Int = 1
    for row in rowStart:height, col in colStart:width
        rectangle[i] = (row, col)
        i += 1
    end
    return rectangle
end
```

Each `rectangle` is represented as a vector of positions (`Vec{Pos}`). It will start at the origin of our coordinate system (`COORD_ORIGIN` - top left corner of our matrix). It will spread through as many `rows` and `columns` as there are needed. Their numbers will be calculated based on the `height` and `width` of the rectangle. Such a rectangle (the one that starts in the coordinate system origin point) is a good start, but to draw a picture we need to be able to place a shape in any location on the canvas.

```
# moves a shape by (nRows, nCols)
function nudge(shape::Vec{Pos}, by::Pos)::Vec{Pos}
    return map(pos -> pos .+ by, shape)
end

# shifts a shape so that its anchor point starts where we want
function shift(shape::Vec{Pos}, anchor::Pos)::Vec{Pos}
```

```
    shift::Pos = anchor .- COORD_ORIGIN
    return nudge(shape, shift)
end

function getRectangle(width::Int, height::Int,
topLeftCorner::Pos)::Vec{Pos}
    return shift(getRectangle(width, height), topLeftCorner)
end
```

So far so good. Time to find a way to add the points that build our shape to the canvas (notice, that we only add the points that are inside of our canvas).

```
function isWithinCanvas(point::Pos, cvs::Matrix{Str}=canvas)::Bool
    nRows, nCols = size(cvs)
    row, col = point
    return (0 < row <= nRows) && (0 < col <= nCols)
end

function addPoints!(shape::Vec{Pos}, color::Symbol,
                    cvs!::Matrix{Str}=canvas)::Nothing
    for pt in shape
        if isWithinCanvas(pt, cvs!)
            cvs![pt...] = getBgColor(color)
        end
    end
    return nothing
end
```

Notice `!` character in `addPoints!`. Per Julia's convention it was added to the name of the function that modifies its contents. However, both `shape` (`Vec{Pos}`) and `cvs` (`Matrix{Str}`) are passed by reference. So a question may arise which one of the two (or maybe both) will get modified. To help with the answer the second parameter was named `cvs!` to emphasize that only it will be modified by the function (the `!` is just a part of the function's parameter's name).

Once we got it we can move to another shape, i.e. a triangle.

```
function getTriangle(height::Int)::Vec{Pos}
    @assert height > 1 "height must be > 1"
    rowStart::Int, colStart::Int = COORD_ORIGIN
    lCol::Int = colStart # 1
```

```
    rCol::Int = colStart # 2
    triangle::Vec{Pos} = []
    for row in rowStart:height
        for col in lCol:rCol
            push!(triangle, (row, col))
        end
        lCol -= 1
        rCol += 1
    end
    return triangle
end

function getTriangle(height::Int, apex::Pos)::Vec{Pos}
    return shift(getTriangle(height), apex)
end
```

Our triangle's top starts with a pixel (`lCol` and `rCol` are initialized with the same value) in the origin of our coordinate system (`COORD_ORIGIN`). Then for each row (`for row in rowStart:height`) we dye each pixel between the left (`lCol`) and right (`rCol`) columns (inclusive-inclusive). The basis of the triangle is increased by one pixel on each side (`lCol -= 1` and `rCol += 1`) with each row we move down. Of note, we could have shortened the above snippet, e.g. by using a C[190]-like chained assignment (`lCol = rCol = colStart` instead of lines `#1` and `#2`). However, the longer version might be clearer and easier to follow in a head.

There's one more shape left, a circle.

```
function getCircle(radius::Int)::Vec{Pos}
    @assert 1 < radius < 6 "radius must be in range [2-5]"
    cols::Vec{Vec{Int}} = [collect((-1-r):(2+r)) for r in 0:(radius-1)]
    cols = [cols..., reverse(cols)...]
    circle::Vec{Pos} = []
    rowStart::Int, _ = COORD_ORIGIN
    for row in rowStart:(radius*2)
        for col in cols[row]
            push!(circle, (row, col))
        end
    end
    return circle
end
```

[190] https://en.wikipedia.org/wiki/C_(programming_language)

```
function getCircle(radius::Int, topCenter::Pos)::Vec{Pos}
    return shift(getCircle(radius), topCenter)
end
```

Here, we use a pattern similar to the one from the triangle. A circle is started in the top row (`rowStart`) with three columns (`collect((-1-r):(2+r))`). With every row down we increase the spread by 1 column in each direction (`r` changes by 1). Once, we are in half of our circle we decrease the number of colored columns. We achieve that by combining the previous `cols` with their `reversed` version (`...` is a splat operator that, unpacks a vector by copying its elements).

Finally, we proceed to create our pixel-art graphics, e.g. by iteratively adding one element at a time with something like:

```
clearCanvas!()
addPoints!(getSomeShape, :someColor)
printCanvas()
```

and

```
clearCanvas!()
addPoints!(getSomeShape, :someColor)
addPoints!(getAnotherShape, :someOtherColor)
printCanvas()
```

Until we reach a satisfactory result with a code snippet similar to the one below:

```
clearCanvas!()
addPoints!(getRectangle(60, 15, (16, 1)), :green) # meadow
addPoints!(getRectangle(60, 15), :blue) # sky
addPoints!(getRectangle(15, 8, (15, 21)), :white) # house walls
addPoints!(getRectangle(6, 6, (17, 28)), :brown) # doors
addPoints!(getRectangle(4, 2, (16, 22)), :darkblue) # window
addPoints!(getRectangle(4, 6, (8, 31)), :black) # chimney
addPoints!(getTriangle(8, (7, 28)), :red) # roof
addPoints!(getCircle(4, (2, 55)), :yellow) # sun
addPoints!(getCircle(3, (4, 7)), :white) # cloud, part 1
addPoints!(getCircle(4, (4, 14)), :white) # cloud, part 2
```

```
addPoints!(getCircle(2, (2, 18)), :white) # cloud, part 3
printCanvas()
```

Tree

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[191](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[192].

Problem

There is this nice little command tree[193]that recursively lists files and sub-directories in a given location. Let's try to get some of that with Julia.

Write a function `printCatalogTree` that displays the contents of a given directory like so (the output doesn't have to be exact):

```
~/Desktop/catalog_x/
|--- catalog_y/
|    |--- catalog_z/
|    |    |--- file_z.txt
|    |--- file_y.txt
|--- file_x.txt

2 directories, 3 files
```

Here, we got three nested catalogs: x, y, and z. Each contains a file of a corresponding name inside of it.

Hint: If you are stuck now, start by reading about Julia's Filesystem[194].

---

[191]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/tree
[192]https://docs.julialang.org/en/v1/stdlib/Pkg/
[193]https://en.wikipedia.org/wiki/Tree_(command)
[194]https://docs.julialang.org/en/v1/base/file/

Solution

Let's start small with an initial definition of `printCatalogTree`.

```
function printCatalogTree(path::Str, pad::Str)::Nothing
    newPad::Str = pad * "    "
    for name in readdir(path)
        newPath::Str = joinpath(path, name)
        if isfile(newPath)
            println(newPad, name)
        else
            println(newPad, name, "/")
            printCatalogTree(newPath, newPad)
        end
    end
    return nothing
end

function printCatalogTree(path::Str)::Nothing
    if !isdir(path)
        println("The path $path does not exist.")
        return nothing
    end
    println(path, "/")
    printCatalogTree(path, "")
    return nothing
end
```

The function is rather simple. We walk through every entry (`for name`) in the examined directory (`path`). As we go `name` is converted to `newPath` which will be examined in a moment. If `newPath` is a file (`isfile`) we just print it, otherwise (`else`) it is a directory and we print it with `"/"` to make it stand out. Moreover, we go inside of it with `printCatalogTree` (recursive call) to print its contents. For every nesting of `printCatalogTree` we update the padding `newPad` by increasing the indentation with `* "    "`.

We conclude with another version of `printCatalogTree`. The method will make the function's invocation slightly easier (it requires only one argument so the user does not have to think what the `pad` should be). Moreover, it handles the possibility that the `path` may not exist (`if !isdir(path)`). Additionally, it will add a header line for us (`println(path, "/")`) in order to display the root directory.

Let’s see how it works (remember to create `catalog_x` with its contents first).

```
printCatalogTree(joinpath(homedir(), "Desktop", "catalog_x"))
```

```
/home/user_name/Desktop/catalog_x/
    catalog_y/
        catalog_z/
            file_z.txt
        file_y.txt
    file_x.txt
```

It looks quite alright. Time to replace the spaces on the left with some guideways that we may follow with our eyes.

```
function printCatalogTree(path::Str, pad::Str)::Nothing
    newPad::Str = pad * "--- "
    for name in readdir(path)
        newPath::Str = joinpath(path, name)
        if isfile(newPath)
            println(newPad, name)
        else
            println(newPad, name, "/")
            printCatalogTree(newPath, pad * "    |")
        end
    end
    return nothing
end

function printCatalogTree(path::Str)::Nothing
    if !isdir(path)
        println("The path $path does not exist.")
        return nothing
    end
    println(path, "/")
    printCatalogTree(path, "|")
    return nothing
end
```

To that end we changed the `pads`. We begin with the first column on the left that should start with the pipe character (`printCatalogTree(path, "|")`). Each line containing a file or directory should continue with hyphens “—” (`newPad::Str = pad * "---"`), whereas a nested directory and its contents should be indented

and start with the pipe character as well (`printCatalogTree(newPath, pad * "    |")`). Let's see did it work as intended.

```
printCatalogTree(joinpath(homedir(), "Desktop", "catalog_x"))
```

```
/home/user_name/Desktop/catalog_x/
|--- catalog_y/
|    |--- catalog_z/
|    |    |--- file_z.txt
|    |--- file_y.txt
|--- file_x.txt
```

Not bad at all. Time to add a summary line.

```
function printCatalogTree!(path::Str, pad::Str,
                           count::Dict{Str, Int})::Nothing
    newPad::Str = pad * "--- "
    for name in readdir(path)
        newPath::Str = joinpath(path, name)
        if isfile(newPath)
            println(newPad, name)
            count["nFiles"] += 1
        else
            println(newPad, name, "/")
            count["nDirs"] += 1
            printCatalogTree!(newPath, pad * "    |", count)
        end
    end
    return nothing
end

function printCatalogTree(path::Str)::Nothing
    if !isdir(path)
        println("The path $path does not exist.")
        return nothing
    end
    println(path, "/")
    count::Dict{Str, Int}= Dict("nDirs" => 0, "nFiles" => 0)
    printCatalogTree!(path, "|", count)
    print("\n", count["nDirs"], " directories, ", count["nFiles"], "
files")
    return nothing
end
```

The necessary data will be collected in a dictionary (`count`). It has two keys: `nDirs` and `nFiles` for the number of directories and files in the

tree, respectively. Remember that in Julia dictionaries are passed by reference so any modification you make to them inside of a function will be visible outside. For that purpose `count` is defined in the outer `printCatalogTree`, whereas the inner `printCatalogTree!` got exclamation point per Julia's convention to mark a function that modifies its arguments. Anyway, the `count` dictionary is updated for every file (`count["nFiles"] += 1`) and directory (`count["nDirs"] += 1`) encountered and a summary line is added at the very bottom of the output (`print("\n", count["nDirs"], " directories, ", count["nFiles"], " files")`). Let's see how it works on a few examples (feel free to recreate mock directory structures as seen here).

First, a simple, already familiar to us, case.

```
printCatalogTree(joinpath(homedir(), "Desktop", "catalog_x"))
```

```
/home/user_name/Desktop/catalog_x/
|--- catalog_y/
|    |--- catalog_z/
|    |    |--- file_z.txt
|    |--- file_y.txt
|--- file_x.txt

2 directories, 3 files
```

Now, a bit more convoluted tree.

```
printCatalogTree(joinpath(homedir(), "Desktop", "catalog_a"))
```

```
/home/user_name/Desktop/catalog_a/
|--- catalog_b/
|    |--- catalog_d/
|    |    |--- file_d.txt
|    |--- file_b1.txt
|    |--- file_b2.txt
|--- catalog_c/
|    |--- catalog_e/
|    |    |--- file_e.txt
|    |--- catalog_f/
|    |    |--- file_f1.txt
|    |    |--- file_f2.txt
|    |    |--- file_f3.txt
```

```
|    |    |--- file_f4.txt
|--- file_a.txt

5 directories, 9 files
```

And an empty directory.

```
printCatalogTree(joinpath(homedir(), "Desktop", "catalog_zzz"))
```

```
/home/user_name/Desktop/catalog_zzz/

0 directories, 0 files
```

OK, that's it. Another tiny utility under our belts.

Format Text

In this chapter I used the following libraries.

```
import Random as Rnd # internal library
```

Still, once you read the problem description you may decide to do otherwise.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[195](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[196].

Problem

Word processing programs[197] offer many text editing functionalities. In a pre-digital era, those tasks had to be done manually. Anyway, let's try to do some typesetting with Julia.

Choose an exemplary text, like `text2beFormatted.txt` from the code snippets[198] or Lorem ipsum from this Wikipedia's page[199]. Next, write a program (a series of functions) that will allow you to:

1. Left align,

```
-----------------------------------------------------------------
|  Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed  |
|  do eiusmod tempor incididunt ut labore et dolore magna        |
|  aliqua. Ut enim ad minim veniam, quis nostrud exercitation    |
|  ullamco laboris nisi ut aliquip ex ea commodo consequat.      |
|  Duis aute irure dolor in reprehenderit in voluptate velit     |
```

[195] https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/format_text

[196] https://docs.julialang.org/en/v1/stdlib/Pkg/

[197] https://en.wikipedia.org/wiki/List_of_word_processor_programs

[198] https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/format_text

[199] https://en.wikipedia.org/wiki/Lorem_ipsum

```
|  esse cillum dolore eu fugiat nulla pariatur. Excepteur sint   |
|  occaecat cupidatat non proident, sunt in culpa qui officia    |
|  deserunt mollit anim id est laborum.                          |
------------------------------------------------------------------
```

2. Right align,

```
------------------------------------------------------------------
|  Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed  |
|        do eiusmod tempor incididunt ut labore et dolore magna  |
|    aliqua. Ut enim ad minim veniam, quis nostrud exercitation  |
|      ullamco laboris nisi ut aliquip ex ea commodo consequat.  |
|     Duis aute irure dolor in reprehenderit in voluptate velit  |
|   esse cillum dolore eu fugiat nulla pariatur. Excepteur sint  |
|    occaecat cupidatat non proident, sunt in culpa qui officia  |
|                          deserunt mollit anim id est laborum.  |
------------------------------------------------------------------
```

3. Center,

```
------------------------------------------------------------------
|  Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed  |
|     do eiusmod tempor incididunt ut labore et dolore magna     |
|   aliqua. Ut enim ad minim veniam, quis nostrud exercitation   |
|    ullamco laboris nisi ut aliquip ex ea commodo consequat.    |
|   Duis aute irure dolor in reprehenderit in voluptate velit    |
|  esse cillum dolore eu fugiat nulla pariatur. Excepteur sint   |
|   occaecat cupidatat non proident, sunt in culpa qui officia   |
|              deserunt mollit anim id est laborum.              |
------------------------------------------------------------------
```

4. Justify,

```
------------------------------------------------------------------
|  Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed  |
|  do eiusmod  tempor  incididunt  ut labore  et  dolore  magna  |
|  aliqua. Ut  enim ad minim  veniam, quis nostrud exercitation  |
|  ullamco  laboris nisi  ut  aliquip ex ea  commodo consequat.  |
|  Duis aute irure  dolor  in  reprehenderit in voluptate velit  |
|  esse cillum dolore eu fugiat nulla pariatur. Excepteur  sint  |
|  occaecat cupidatat  non proident, sunt in culpa  qui officia  |
|  deserunt mollit anim id est laborum.                          |
------------------------------------------------------------------
```

5. Justify in a double column layout

```
-------------------------------------------------------------------
|  Lorem ipsum dolor sit  amet,    aute    irure    dolor    in  |
|  consectetur adipiscing elit,    reprehenderit  in  voluptate  |
|  sed   do    eiusmod   tempor    velit esse  cillum dolore eu  |
|  incididunt  ut   labore   et    fugiat    nulla    pariatur.  |
|  dolore magna aliqua. Ut enim    Excepteur   sint    occaecat  |
|  ad    minim   veniam,   quis    cupidatat non proident, sunt  |
|  nostrud exercitation ullamco    in    culpa    qui   officia  |
|  laboris  nisi  ut aliquip ex    deserunt  mollit anim id est  |
|  ea  commodo consequat.  Duis    laborum.                      |
-------------------------------------------------------------------
```

the text (the borders are optional).

For simplicity, you may assume the text to be composed of ASCII encoded symbols[200]with the words composed of, let's say, up to 10-15 letters and separated by an unspecified number of whitespace characters[201]. Feel free to adjust the task to your programming experience and do as many tasks as you deem suitable.

Solution

Before we begin, let's define a few const[202]ants that will be used throughout the program.

```
const PAD = " "
const COL_SEP = PAD ^ 4
const MAX_LINE_LEN = 60
```

Next, our first step in formatting a paragraph will be to break it into lines, each with the length smaller than or equal to some target value.

```
function getLines(txt::Str, targetLineLen::Int=MAX_LINE_LEN)::Vec{Str}
    @assert 19 < targetLineLen < 61 "targetLineLen must be in range
[20-60]"
    words::Vec{Str} = split(txt)
    lines::Vec{Str} = []
    curLine::Str = ""
    difference::Int = 0
    for word in words
```

[200]https://en.wikipedia.org/wiki/ASCII
[201]https://en.wikipedia.org/wiki/Whitespace_character
[202]https://docs.julialang.org/en/v1/base/base/#const

```
        difference = targetLineLen - length(curLine) - length(word)
        if difference >= 0
            curLine *= word * PAD
        else
            push!(lines, strip(curLine))
            curLine = word * PAD
        end
    end
    push!(lines, strip(curLine))
    return lines
end
```

For that we break our text (`txt`) into `words` with the split[203]function. Then, `for` each `word` we calculate the `difference` between our desired line length (`targetLineLen`) and the current line length (`length(curLine)`) plus the length of the word (`length(word)`) we want to add to that line. If we still got room for one more word (`if difference >= 0`) then we just add it with a padding on the right side (`curLine *= word * PAD`). Otherwise (`else`), we add `curLine` to the vector of `lines` with `push!` and make our `word` the beginning of a new line (`curLine = word * PAD`). Notice, that before `push`ing the old line to the collection, first, we `strip`ped it from any possible extra spaces on the edges. Afterwards (`end` of `for`), we `push` the last line to `lines` and return the latter from inside the function.

Now, for left-, right- and center alignment, each line will have to be padded with space characters (`PAD`) placed on the right, left, and both sides, respectively. For that we need to know the difference between the number of characters in our line and its target length. Moreover, we need a padding function that we will name creatively as `padLine`. Here, we went with the * operator that glues two strings together and the ^ symbol that repeats the string on its left the number of times on its right (remember about the operator precedence from mathematics). Alternatively, we could have used the built in @sprintf[204] (e.g. `@sprintf("%60s", "xxx")`/`@sprintf("%-60s", "xxx")` to get the

[203]https://docs.julialang.org/en/v1/base/strings/#Base.split
[204]https://docs.julialang.org/en/v1/stdlib/Printf/#Printf.@sprintf
[205]https://docs.julialang.org/en/v1/base/strings/#Base.lpad

"xxx" right/left justified for us). Lpad[205] and rpad[206] were also a viable option.

```
function getLenDiffs(lines::Vec{Str},
                     targetLineLen::Int=MAX_LINE_LEN)::Vec{Int}
    return targetLineLen .- map(length, lines)
end

function padLine(line::Str, lPadLen::Int, rPadLen::Int,
                 lPad::Str=PAD, rPad::Str=PAD)::Str
    @assert lPadLen >= 0 && rPadLen >= 0 "padding lengths must be >= 0"
    return lPad ^ lPadLen * line * rPad ^ rPadLen
end
```

Once we got it, padding lines should be a breeze.

```
function getPaddedLines(lines::Vec{Str},
                         lPadsLens::Vec{Int},
rPadsLens::Vec{Int})::Vec{Str}
    @assert(length(lines) == length(lPadsLens) == length(rPadsLens),
            "all vectors must be of equal lengths")
    return map(padLine, lines, lPadsLens, rPadsLens)
end
```

Here, similarly to Section 2.2 , we use map[207], which applies a function (`padLine`) to every consecutive elements of `lines`, `lPadsLens` and `rPadsLens` in turn. So it goes like: `padLine(lines[1], lPadsLens[1], rPadsLens[1])` and `padLine(lines[2], lPadsLens[2], rPadsLens[2])`, etc., and collects the results into a vector.

Now, the formatting reduces down to obtaining the lines, and calculating the diffs, which we do on the fly with this code snippet (`div(x, y)` divides `x` by `y` and returns an integer by dropping fractional part when necessary).

```
function getLeftAlignedLines(txt::Str,
                              targetLineLen::Int=MAX_LINE_LEN)::Vec{Str}
    lines::Vec{Str} = getLines(txt, targetLineLen)
    rPadsLens::Vec{Int} = getLenDiffs(lines, targetLineLen)
    return getPaddedLines(lines, zeros(Int, length(lines)), rPadsLens)
```

[206] https://docs.julialang.org/en/v1/base/strings/#Base.rpad
[207] https://docs.julialang.org/en/v1/base/collections/#Base.map

```
end

function getRightAlignedLines(txt::Str,
                              targetLineLen::Int=MAX_LINE_LEN)::Vec{Str}
    lines::Vec{Str} = getLines(txt, targetLineLen)
    lPadsLens::Vec{Int} = getLenDiffs(lines, targetLineLen)
    return getPaddedLines(lines, lPadsLens, zeros(Int, length(lines)))
end

function getCenteredLines(txt::Str,
                          targetLineLen::Int=MAX_LINE_LEN)::Vec{Str}
    lines::Vec{Str} = getLines(txt, targetLineLen)
    diffs::Vec{Int} = getLenDiffs(lines, targetLineLen)
    lPadLens::Vec{Int} = div.(diffs, 2)
    rPadLens::Vec{Int} = diffs .- lPadLens
    return getPaddedLines(lines, lPadLens, rPadLens)
end

function printLines(lines::Vec{Str})::Nothing
    join(lines, "\n") |> print
    return nothing
end
```

Go ahead, test it out (e.g. `getCenteredLines(lorem) |> printLines`) and see how it works.

OK, time for a function that will justify our line (`justifyLine`). Here, our approach will be slightly different. First, we'll break the line into words (with `split`). Then, we'll figure out how many regular (`nSpacesBtwnWords`) spaces and extra spaces (`nExtraSpaces`) between the words we need. Finally, we'll place the extra spaces in random places (with `getSample`) between the words (with `intercalate`).

```
# draws n random elements from v (without replacement)
function getSample(v::Vec{A}, n::Int)::Vec{A} where A
    @assert 0 <= n <= length(v) "n must be in range [0-length(v)]"
    return Rnd.shuffle(v)[1:n]
end

function intercalate(v1::Vec{Str}, v2::Vec{Str})::Str
    @assert(length(v1) == (length(v2)+1),
            "length(v1) must be equal length(v2)+1")
    return join(map(*, v1, v2)) * v1[end]
end

function justifyLine(line::Str, lastLine::Bool=false,
```

```
                     targetLineLen::Int=MAX_LINE_LEN)::Str
    words::Vec{Str} = split(line)
    if  length(words) < 2 || lastLine
        return padLine(line, 0, targetLineLen - length(line))
    end
    nSpacesBtwnWords::Int = length(words) - 1
    nSpacesTotal::Int = targetLineLen - sum(map(length, words))
    spaceBtwnWordsLen::Int = div(nSpacesTotal, nSpacesBtwnWords)
    nExtraSpaces::Int = nSpacesTotal - nSpacesBtwnWords *
spaceBtwnWordsLen
    spaces::Vec{Str} = fill(PAD ^ spaceBtwnWordsLen, nSpacesBtwnWords)
    inds::Vec{Int} = getSample(collect(eachindex(spaces)), nExtraSpaces)
    spaces[inds] .*= PAD
    return intercalate(words, spaces)
end
```

Let's briefly discuss some of the more interesting parts of the code snippet. We start by determining how many spaces between the words there are (`nSpacesBtwnWords`) and how many spaces in total we need in order to reach our `targetLineLen` (`nSpacesTotal`). In the perfect world, `nSpacesTotal` should divide by `nSpacesBtwnWords` evenly (`spaceBtwnWordsLen` should be an integer). To help that happen we use `div` (it drops the fractional part). Moreover, we also account for the possible extra spaces needed (`nExtraSpaces`, when the dropped fractional part was greater than 0). Once we got it, we get a vector of regular spaces between the words and place it in `spaces`. Then, we draw random indices (`inds`) from `spaces` to which we will add a single extra space (`PAD`). Notice that `spaces[inds] .= PAD` would replace every element of `spaces` (indicated by `inds`) with `PAD`. Instead, `spaces[inds] .*= PAD` will take every element and update it (*=) with `PAD`, which in this case means just appending (*) `PAD` (a string) to the string that was previously in an element of `spaces`. Finally, we intercalate `words` in a `line` with `spaces` (regular and extra), which we `return`. And voila, all that's left to do is to justify every line

```
function getJustifiedLines(txt::Str,
                           targetLineLen::Int=MAX_LINE_LEN)::Vec{Str}
    lines::Vec{Str} = getLines(txt, targetLineLen)
    return map(line -> justifyLine(
        line, line == lines[end], targetLineLen), lines)
end
```

and test it out (`getJustifiedLines(lorem) |> printLines`).

As for the double column justified layout, it's pretty straightforward. We'll get a single justified column (that is roughly half the width of `MAX_LINE_LEN`), split it approximately in half and glue together lines from adjacent columns. Let's get into it.

```
function connectColumns(col1lines::Vec{Str},
col2lines::Vec{Str})::Vec{Str}
    @assert(length(col1lines) >= length(col2lines),
            "col1lines must have >= elements than col2lines")
    result::Vec{Str} = fill("", length(col1lines))
    emptyColPad = padLine("", 0, length(col1lines[1]))
    for i in eachindex(col1lines)
        result[i] = string(col1lines[i], COL_SEP,
                           get(col2lines, i, emptyColPad))
    end
    return result
end

function getDoubleColumn(txt::Str,
                         targetLineLen::Int=MAX_LINE_LEN)::Vec{Str}
    @assert 19 < targetLineLen < 61 "targetLineLen must be in range
[20-60]"
    lines::Vec{Str} = getJustifiedLines(
        txt, div(targetLineLen, 2) - div(length(COL_SEP), 2), ) # 2 -
nCols
    midPoint::Int = ceil(Int, length(lines)/2)
    return connectColumns(lines[1:midPoint], lines[(midPoint+1):end])
end
```

Of note, `connectColumns` walks through every index in `col1lines` (`eachindex(col1lines)`) and glues together the columns with the `string` function. The outcome of such string concatenation is put into `result[i]`. Splitting a long thin column in half may result in a columns of a slightly different lengths. Therefore, we cannot just use a regular indexing on `col2lines` (because if the element is not there we'll get an error). Instead, we use the `get` function that we encountered while working with dictionaries[208]. Likewise, here we also use a default value (`emptyColPad`) that gets returned if an element at a given index does not exist. It seems that we're done with the chapter's

[208] https://b-lukaszuk.github.io/RJ_BS_eng/julia_language_decision_making.html#sec:julia_language_dictionaries

task. Go ahead, test the last function (`getDoubleColumn(lorem) |> printLines`).

For a final challenge (a cherry on the top) add borders to the printout (or use the function `addBorder` from the code snippets[209]). This will allow to better visualize the correctness of your padding.

[209] https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/format_text

Roman Numerals

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[210](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[211].

Problem

Every now and then we encounter some Roman numerals written on an old building's wall or a book's page. Your job is to refresh your knowledge about the numerals, e.g. by reading this Wikipedia's entry[212], and write a two way converter, for instance in the form of `getRoman(arabic::Int)::Str` and `getArabic(roman::Str)::Int` functions. Below you will find two sets of parallel numbers (based on the Wikipedia's page) so that you can test your results (your solution should work correctly for the numbers in range [1-3999]).

```
arabicTest = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
              11, 12, 39, 246, 789, 2421, 160, 207, 1009, 1066,
              3999, 1776, 1918, 1944, 2025,
              1900, 1912]
romanTest = ["I", "II", "III", "IV", "V",
             "VI", "VII", "VIII", "IX", "X",
             "XI", "XII", "XXXIX", "CCXLVI", "DCCLXXXIX",
             "MMCDXXI", "CLX", "CCVII", "MIX", "MLXVI",
             "MMMCMXCIX", "MDCCLXXVI", "MCMXVIII", "MCMXLIV", "MMXXV",
             "MCM", "MCMXII"]
```

Good luck.

[210]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/roman_numerals

[211]https://docs.julialang.org/en/v1/stdlib/Pkg/

[212]https://en.wikipedia.org/wiki/Roman_numerals

Solution

Let's start with a simple mapping between some key Roman numerals and their Arabic counterparts.

```
const ROMAN_2_ARABIC = [("M", 1000), ("CM", 900),
                ("D", 500), ("CD", 400),
                ("C", 100), ("XC", 90),
                ("L", 50), ("XL", 40),
                ("X", 10), ("IX", 9), ("V", 5),
                ("IV", 4), ("I", 1)]
```

The mapping is defined with (`const`) keyword to signal that we do not wish to change it throughout the program execution. Moreover, we used a vector of tuples, not a dictionary, since we want to preserve the (descending) order of the pairs of values. Notice, that we inclued the key landmarks of subtractive notation (e.g. `("CM", 900)` or `("IV", 4)`).

> Note. Using `const` with mutable containers like vectors or dictionaries allows to change their contents later on, e.g., with `push!`. So the `const` used here is more like a convention, a signal that we do not plan to change the containers in the future. If we really wanted an immutable container then we should consider a(n) (immutable) tuple. Anyway, some programming languages suggest that `const` names should be declared using all uppercase characters to make them stand out. Here, I follow this convention.

Now, we are ready to take the next step.

```
function getRoman(arabic::Int)::Str
    @assert 0 < arabic < 4000
    roman::Str = ""
    for (r, a) in ROMAN_2_ARABIC
        while(arabic >= a)
            roman *= r
            arabic -= a
        end
    end
    return roman
end
```

We will build our Roman numeral bit by bit starting from an empty string (`roman::Str = ""`). For that we traverse all our Roman and Arabic landmarks (`for (r, a) in ROMAN_2_ARABIC`). For each of them (starting from the highest number), we check if the currently examined Arabic landmark (`a`) is lower than the Arabic number we got to translate (`arabic`). As long as it is (`while(arabic >= a)`) we append the parallel Roman landmark (`r`) to our solution (`roman *= r`) and subtract the Arabic landmark (`a`) from our Arabic number (`arabic -= a`). Once we are done we return the result.

> Note. To better understand the above code you may read about the `while` loop in the docs[213]and use show[214]macro to have a sneak peak how the variables in the loop change. If you ever accidentally write an infinite `while` loop you may try to break the program execution by pressing Ctrl+C[215]in your terminal/ command line.

Time for a minitest (go ahead pick a number and, in your head or on a piece of paper, follow the function's execution).

```
getRoman.(1:10)
```

```
["I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX", "X"]
```

OK, time for a bigger test.

```
getRoman.(arabicTest) == romanTest
```

true

Looks alright.

[213]https://docs.julialang.org/en/v1/base/base/#while
[214]https://docs.julialang.org/en/v1/base/base/#Base.@show
[215]https://en.wikipedia.org/wiki/Control-C

Time to write our `getArabic` function. For that we will have to break a Roman numeral into tokens (from left to right) that we will use to build up an Arabic number.

```
const ROMAN_TOKENS = map(first, ROMAN_2_ARABIC)
# equivalent to
# const ROMAN_TOKENS = map(tuple -> tuple[1], ROMAN_2_ARABIC)

function getTokenAndRest(roman::Str)::Tuple{Str, Str}
    if length(roman) <= 1
        return (roman, "")
    elseif roman[1:2] in ROMAN_TOKENS
        return (roman[1:2], string(roman[3:end]))
    else
        return (string(roman[1]), string(roman[2:end]))
    end
end

function getTokens(roman::Str)::Vec{Str}
    curToken::Str = ""
    allTokens::Vec{Str} = []
    while (roman != "")
        curToken, roman = getTokenAndRest(roman)
        push!(allTokens, curToken)
    end
    return allTokens
end
```

First, we extract `ROMAN_TOKENS` with `map` and first[216]. Next, we use `getTokenAndRest` to split a Roman numeral into a key token (first on the left, either one or two characters long) and the rest of the numeral. The `string(sth)` part makes sure we always return a string and not a character. Anyway, based on it (`getTokenAndRest`) we split a Roman numeral into a vector of tokens with `getTokens`. Now, we are ready to write our `getArabic`.

```
function getVal(lookup::Vec{Tuple{Str, Int}}, key::Str,
default::Int)::Int
    for (k, v) in lookup
        if k == key
            return v
        end
    end
```

[216]https://docs.julialang.org/en/v1/base/collections/#Base.first

```
    return default
end

function getArabic(roman::Str)::Int
    tokens::Vec{Str} = getTokens(roman)
    sum::Int = 0
    for curToken in tokens
        sum += getVal(ROMAN_2_ARABIC, curToken, 0)
    end
    return sum
end
```

Notice, that before we moved to `getArabic` we wrote `getVal` that will provide us with an Arabic number for a given Roman numeral token. The above was not strictly necessary, as we could have just use the built-in get[217]for that purpose (e.g. with `get(Dict(ROMAN_2_ARABIC), key, default)`). Anyway, to get the Arabic number for a given Roman numeral (`roman`), first we split it into a vector of `tokens` and traverse them one by one (`curToken`) with the `for` loop. We translate `curToken` to an Arabic numeral and add it to `sum` which we return once we are done.

Again, let's go with a minitest (go ahead pick a number and, in your head or on a piece of paper, follow the function's execution).

```
getArabic.(["I", "II", "III", "IV", "V", "VI", "VII", "VIII"])
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

Finally, a bigger test.

```
getArabic.(romanTest) == arabicTest
```

true

And another one.

```
getArabic.(getRoman.(arabicTest)) == arabicTest
```

[217] https://docs.julialang.org/en/v1/base/collections/#Base.get

true

I don’t know about you, but to me the results are satisfactory.

Cheque

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[218](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[219].

Problem

Every now and then it comes in handy to be able to write down an English numeral[220]with the words. Case in point would be to write a sum of money on a cheque or to display it in a finance (perhaps bank) app. So here is a task for you. Write a program in Julia that for any number let's say in the range of 1 to 999,999 returns its transcription with words. You may handle only integers.

Solution

We begin by defining a few constants that will be useful later on.

```
const UNITS_AND_TEENS = Dict(1 => "one",
                             2 => "two", 3 => "three", 4 => "four",
                             5 => "five", 6 => "six", 7 => "seven",
                             8 => "eight", 9 => "nine", 10 => "ten",
                             11 => "eleven", 12 => "twelve",
                             13 => "thirteen", 14 => "fourteen",
                             15 => "fifteen", 16 => "sixteen",
                             17 => "seventeen", 18 => "eighteen",
                             19 => "nineteen")

const TENS = Dict(2 => "twenty", 3 => "thrity",
                  4 => "forty", 5 => "fifty", 6 => "sixty",
                  7 => "seventy", 8 => "eigthy", 9 => "ninety")
```

[218]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/cheque
[219]https://docs.julialang.org/en/v1/stdlib/Pkg/
[220]https://en.wikipedia.org/wiki/English_numerals

> Note. Using `const` with mutable containers like vectors or dictionaries allows to change their contents later on, e.g., with `push!`. So the `const` used here is more like a convention, a signal that we do not plan to change the containers in the future. If we really wanted an immutable container then we should consider a(n) (immutable) tuple. Anyway, some programming languages suggest that `const` names should be declared using all uppercase characters to make them stand out. Here, I follow this convention.

The above are just mappings between the necessary basic key ingredients of our number soup. Let's use them to get the transcript for numbers in the range of 1 to 99.

```
function getEngNumeralUpto99(n::Int)::Str
    @assert 1 <= n <= 99 "n must be in range [1-99]"
    if n < 20
        return UNITS_AND_TEENS[n]
    end
    t::Int, u::Int = divrem(n, 10) # t - tens, u - units
    result::Str = TENS[t]
    if u != 0
        result *= "-" * UNITS_AND_TEENS[u]
    end
    return result
end
```

Whenever a number (`n`) is below 20 (`if n < 20`) we just return its representation from the `UNITS_AND_TEENS` dictionary. Alternatively, for `n` in the `TENS`, first we obtain the tens (`t`) and units (`u`) part. divrem[221]is a combination of `div` (an integer division) and `rem` (a reminder after the division). We use it, to obtain the transcript for the TENS (`TENS[t]`). It becomes our `result` to which we attach the transcript for units separated by a hyphen (`"-" * UNITS_AND_TEENS[u]`), but only if the unit is different than zero (`if u != 0`).

Time for a test ride.

[221]https://docs.julialang.org/en/v1/base/math/#Base.divrem

```
(
    getEngNumeralUpto99.([5, 9, 11, 13, 20, 21]),
    getEngNumeralUpto99.([25, 32, 58, 64, 66]),
    getEngNumeralUpto99.([79, 83, 95, 99])
)
```

```
(["five", "nine", "eleven", "thirteen", "twenty", "twenty-one"],
["twenty-five", "thrity-two", "fifty-eight", "sixty-four", "sixty-six"],
["seventy-nine", "eigthy-three", "ninety-five", "ninety-nine"])
```

Good. It appears we got that one out of our way. Let's move on to something bigger.

```
function getEngNumeralUpto999(n::Int)::Str
    @assert 1 <= n <= 999 "n must be in range [1-999]"
    if n < 100
        return getEngNumeralUpto99(n)
    end
    h::Int, t::Int = divrem(n, 100) # h - hundreds, t - tens
    result::Str = getEngNumeralUpto99(h) * " hundred"
    if t != 0
        result *= " and " * getEngNumeralUpto99(t)
    end
    return result
end
```

This time we deal with all the integers below one thousand. The previously defined `getEngNumeralUpto99` is an important building block of that new function. If a number (`n`) is smaller than 100 (`if n < 100`) we just transcribe it as we did in the previous code snippet (`return getEngNumeralUpto99(n)`). Otherwise we split it into hundreds (`h`) and TENS part (`t`, actually it may also contain units and teens). We start building our `result` by transcribing the hundreds part (`result::Str = getEngNumeralUpto99(n) * " hundred"`). When appropriate (`t != 0`) we append the transcript for the TENS separated with `"and"` (British English convention) to our `result`.

Time for a test.

```
(
    getEngNumeralUpto999.([9, 13, 20, 66]),
```

```
    getEngNumeralUpto999.([101, 109, 110]),
    getEngNumeralUpto999.([320, 400, 500])
)
```

```
(["nine", "thirteen", "twenty", "sixty-six"],
["one hundred and one", "one hundred and nine", "one hundred and ten"],
["three hundred and twenty", "four hundred", "five hundred"])
```

Since the previous part was so easy, there's not reason to linger. Time for our final leap.

```
function getEngNumeralBelow1M(n::Int)::Str
    @assert 1 <= n <= 999_999 "n must be in range [1-999,999]"
    if n < 1000
        return getEngNumeralUpto999(n)
    end
    t::Int, h::Int = divrem(n, 1000) # t - thousands, h - hundreds
    result::Str = getEngNumeralUpto999(t) * " thousand"
    if h == 0
        return result
    elseif h < 100
        result *= " and "
    else
        result *= ", "
    end
    result *= getEngNumeralUpto999(h)
    return result
end
```

This time we also use our previously defined function (`getEngNumeralUpto999`) as an integral part of the bigger, more general solution. When a number (`n`) is small (`if n < 1000`) we write it down as we used to (`return getEngNumeralUpto999(n)`). Otherwise, we split it (`n`) to the thousands (`t`) and the hundreds (`h`) parts. Next, we build our `result` for the thousands (`getEngNumeralUpto999(t) * " thousand`). When the hundreds part is 0 (`if h == 0`) we just return our `result`. If the hundreds part is small (`h < 100`) we add the conjunction `" and "`, otherwise (large hundreds part), we separate the following words with `", "`. Once again, we finish by using `getEngNumeralUpto999(h)` to transcribe the remaining part.

Let's see our creation at work.

```
(
    getEngNumeralBelow1M.([5, 9, 13, 21, 40, 95]),
    getEngNumeralBelow1M.([101, 320, 500]),
    getEngNumeralBelow1M.([1_800]),
    getEngNumeralBelow1M.([96_779]),
    getEngNumeralBelow1M.([180_000]),
    getEngNumeralBelow1M.([889_308])
)
```

```
(["five", "nine", "thirteen", "twenty-one", "forty", "ninety-five"],
["one hundred and one", "three hundred and twenty", "five hundred"],
["one thousand, eight hundred"],
["ninety-six thousand, seven hundred and seventy-nine"],
["one hundred and eigthy thousand"],
["eight hundred and eigthy-nine thousand, three hundred and eight"])
```

Mission, completed. We wrote three functions that allow us to write down any number we want in the range [1-999,999]. Still, you could argue that there is some code duplication. If that bothers you, you may try to shorten the solution to something like:

```
function getEngNumeral(n::Int)::Str # uses recursion
    @assert 0 < n < 1e6 "n must be in range (0-1e6)"
    major::Int, minor::Int = 0, 0
    if n < 20
        return UNITS_AND_TEENS[n]
    elseif n < 100
        major, minor = divrem(n, 10)
        return TENS[major] * (
            minor == 0 ? "" : "-" * UNITS_AND_TEENS[minor]
        )
    elseif n < 1000
        major, minor = divrem(n, 100)
        return getEngNumeral(major) * " hundred" * (
            minor == 0 ? "" : " and " * getEngNumeral(minor)
        )
    else # < 1e6 due to @assert above
        major, minor = divrem(n, 1_000)
        return getEngNumeral(major) * " thousand" * (
            minor == 0 ? "" :
            minor < 100 ? " and " * getEngNumeral(minor) :
            ", " * getEngNumeral(minor)
        )
```

```
    end
end
```

```
getEngNumeral (generic function with 1 method)
```

This allowed us to reduce the number of lines of code roughly by half, while maintaining the functionality.

```
(
    getEngNumeral.([5, 9, 13, 21, 40, 95]),
    getEngNumeral.([101, 320, 500]),
    getEngNumeral.([1_800]),
    getEngNumeral.([96_779]),
    getEngNumeral.([180_000]),
    getEngNumeral.([889_308])
)
```

```
(["five", "nine", "thirteen", "twenty-one", "forty", "ninety-five"],
["one hundred and one", "three hundred and twenty", "five hundred"],
["one thousand, eight hundred"],
["ninety-six thousand, seven hundred and seventy-nine"],
["one hundred and eigthy thousand"],
["eight hundred and eigthy-nine thousand, three hundred and eight"])
```

Sort

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[222](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[223].

Problem

The Julia's built-in sort[224]function is a pretty useful beast. Let's try to replicate some of its functionality.

Read about different sorting algorithms[225]and implement one or two of them. As a minimum requirement your sorting function/s should correctly sort a vector of numbers (integers and floats), e.g.

```
yourSortingFn([47, 15, 23, 99, 4])
```

Should return:

```
[4, 15, 23, 47, 99]
```

If you want to make it more challenging, try to sort the numbers from 1 to 10 in alphabetical order, e.g.

```
yourSortingFn([1, 11, 6], possibleOtherArgs)
```

should return:

[222]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/sort
[223]https://docs.julialang.org/en/v1/stdlib/Pkg/
[224]https://docs.julialang.org/en/v1/base/sort/#Base.sort
[225]https://en.wikipedia.org/wiki/Sorting_algorithm#Comparison_of_algorithms

```
[11, 1, 6]
```

Since “eleven” comes before “one” and before “six” (“e” < “o” < “s”).

Solution

For our first attempt we’ll go with something simple like the bubble sort[226]algorithm.

```
# sorts in ascending order
function bs(v::Vec{A})::Vec{A} where A<:Union{Flt, Int}
    result::Vec{A} = copy(v)
    swapped::Bool = true
    while swapped
        swapped = false
        for i in eachindex(result)[2:end]
            if result[i-1] > result[i]
                result[i-1], result[i] = result[i], result[i-1]
                swapped = true
            end
        end
    end
    return result
end
```

Here, we implemented the algorithm for a vector that contains the elements of type `A`. In this case it is just a sub-type (`<:`) of `Flt` (an alias for `Float64`) or `Int` (`Union{Flt, Int}`). We started by `copy`ing the original vector to `result` and setting a variable named `swapped`. We will stop our sorting algorithm once there was no swap during the previous traversal of the whole vector (`swapped` in `while` is `false` but may be reset to `true` in the `for` loop). We search through the vector starting from its second element (`i in eachindex(result)[2:end]`). Every time we compare the two nearby elements (`result[i-1]` vs. `result[i]`). If the elements aren’t in a desired order (`if result[i-1] > result[i]`) we just swap them with each other (`result[i-1], result[i] = result[i], result[i-1]`) and set the `swapped` flag to `true`. Once we finish we return the result.

Let’s see how we did.

[226]https://en.wikipedia.org/wiki/Bubble_sort

```
[47, 15, 23, 99, 4] |> bs,
[47.3, 23.2, 47.2] |> bs
```

```
([4, 15, 23, 47, 99], [23.2, 47.2, 47.3])
```

If you want to better visualize how the algorithm works, you may place `println` or @show[227] constructs in a few places inside the function, e.g.

```
function bs(v::Vec{A})::Vec{A} where A<:Union{Flt, Int}
    result::Vec{A} = copy(v)
    swapped::Bool = true
    i::Int = 0
    while swapped
        i += 1
        @show i
        swapped = false
        for i in eachindex(result)[2:end]
            if result[i-1] > result[i]
                result[i-1], result[i] = result[i], result[i-1]
                swapped = true
            end
            @show result, swapped
        end
    end
    return result
end
```

Which prints:

```
[0.75, 0.25, 0.5] |> bs

# printout
i = 1
(result, swapped) = ([0.25, 0.75, 0.5], true)
(result, swapped) = ([0.25, 0.5, 0.75], true)
i = 2
(result, swapped) = ([0.25, 0.5, 0.75], false)
(result, swapped) = ([0.25, 0.5, 0.75], false)
```

[227] https://docs.julialang.org/en/v1/base/base/#Base.@show
[228] https://en.wikipedia.org/wiki/Quicksort

For our next try we will go with the quicksort[228]algorithm. Here, I'll try to implement it from memory based on the algorithm I once saw in "Learn You a Haskell for Great Good!" by Miran Lipovača. Of course, I'll try to adjust it for Julia syntax.

```
# sorts in ascending order
function qs(v::Vec{Int})::Vec{Int}
    if isempty(v)
        return []
    else
        head, tail... = v
        # or: head::Int, tail::Vec{Int}... = v
        smallerElts::Vec{Int} = filter((<)(head), tail)
        greaterEqElts::Vec{Int} = filter((>=)(head), tail)
        return [qs(smallerElts); head; qs(greaterEqElts)]
    end
end
```

To do so we choose a so called pivot element (`head`) that for simplicity is always the first element in the vector (`v`). Next, we take the remaining part of the vector (`tail`). The `head, tail... = v` is a destructuring assignment[229]that puts the first elt of `v` to `head` variable and all the remaining elements to `tail` (if `v` is made of one element, then `tail` is `[]`). Anyway, we separate `tail` elements into the ones that are smaller (`smallerElts`) and greater than or equal to (`greaterEqElts`) our pivot element (`head`). The above is done with `filter` and partial function application[230]. Once we got it we use recursion (e.g. `qs(unsorted_smaller_elts)` in `return`) and vector concatenation (`[vector_or_elt; vector_or_elt; vector_or_elt]`).

Let's see how it works.

```
[47, 15, 23, 99, 4] |> qs
```

```
[4, 15, 23, 47, 99]
```

[229]https://docs.julialang.org/en/v1/manual/functions/#destructuring-assignment
[230]https://bkamins.github.io/julialang/2024/02/23/fix.html

Looks good. OK, in case the above was too great of a squeeze, let's try to make it a bit less functional[231](Haskell is a purely functional programming language) and a bit more imperative[232].

```
function qs(v::Vec{Int})::Vec{Int}
    if isempty(v)
        return []
    else
        firstElt::Int = v[1]
        otherElts::Vec{Int} = v[2:end]
        smallerElts::Vec{Int} = filter(elt -> elt < firstElt, otherElts)
        greaterEqElts::Vec{Int} = filter(elt -> elt >= firstElt,
otherElts)
        return [qs(smallerElts); firstElt; qs(greaterEqElts)]
    end
end
```

Once again, we choose a so called pivot element (`firstElt`) that for simplicity is always the first element of a vector. Next, we take the remaining elements (`otherElts`) and separate them into the elements that are smaller (`smallerElts`) and greater than or equal to (`greaterEqElts`) our pivot element (`firstElt`). The above is done with `filter` and an anonymous function[233]. Once we got it we use recursion (e.g. `qs(unsorted_smaller_elts)` in `return`) and vector concatenation (`[vector_or_elt; vector_or_elt; vector_or_elt]`).

Just a small test to make sure it still works as intended.

```
[47, 15, 23, 99, 4] |> qs
```

```
[4, 15, 23, 47, 99]
```

Flawless victory, but we are still unable to sort the numbers in alphabetical order. Let's fix that by modifying our `qs`.

```
# by - fn(A) -> B; is applied to every elt of v before sorting
# lt - fn(B, B) -> Bool; compares pairs of elts after 'by' was applied
```

[231]https://en.wikipedia.org/wiki/Functional_programming
[232]https://en.wikipedia.org/wiki/Imperative_programming
[233]https://docs.julialang.org/en/v1/manual/functions/#man-anonymous-functions

```
function qs(v::Vec{A},
            by::Function=identity,
            lt::Function=<)::Vec{A} where A
    if isempty(v)
        return []
    else
        firstElt::A = v[1]
        otherElts::Vec{A} = v[2:end]
        smallerElts::Vec{A} = filter(
            elt -> lt(by(elt), by(firstElt)),
            otherElts
        )
        greaterEqElts::Vec{A} = filter(
            elt -> !lt(by(elt), by(firstElt)),
            otherElts
        )
        return [
            qs(smallerElts, by, lt);
            firstElt;
            qs(greaterEqElts, by, lt)
        ]
    end
end
```

Here, we added two more positional arguments to `qs`: 1) `by` which is a function applied to every element of the vector (`v`) before the sorting; 2) `lt` - a comparator function (`lt` - less than, `!lt` - negation of `lt`) that compares our pivot element (`firstElt`) with all the other elements (`otherElts`). The comparison occurs after `by` was applied to all the elements of `v` (`by(elt)` and `by(firstElt)` in `filter`). The names `by` and `lt` were inspired by the names of the keyword arguments[234]in Base.sort[235]. The default function for `by` is identity[236](returns its argument unchanged) and the default function for `lt` is Base.:<[237].

Let's see how we did:

```
[0.75, 0.25, 0.5] |> qs,
['c', 'b', 'a', 'd'] |> qs
```

[234]https://docs.julialang.org/en/v1/devdocs/functions/#Keyword-arguments
[235]https://docs.julialang.org/en/v1/base/sort/#Base.sort
[236]https://docs.julialang.org/en/v1/base/base/#Base.identity
[237]https://docs.julialang.org/en/v1/base/math/#Base.:%3C

```
([0.25, 0.5, 0.75], ['a', 'b', 'c', 'd'])
```

And now for the alphabetical sort (`getEngNumeral` was a solution for the problem in Section 25.1 . Here, we passed it without `by=` because in `qs` it is a positional argument).

```
qs(collect(1:10), getEngNumeral)
```

```
[8, 5, 4, 9, 1, 7, 6, 10, 3, 2]
```

The above should work similar to the built-in `sort` (here we use `by=getEngNumeral` because that's how you pass keyword arguments to a function).

```
qs([0.75, 0.25, 0.5]) == sort([0.75, 0.25, 0.5]),
qs(['c', 'b', 'a', 'd']) == sort(['c', 'b', 'a', 'd']),
qs(collect(1:10), getEngNumeral) == sort(1:10, by=getEngNumeral)
```

```
(true, true, true)
```

Regex

In this chapter I used the following libraries.

```
import Random as Rnd # internal library
```

Still, once you read the problem description you may decide to do otherwise.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[238](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[239].

Problem

Regex Intro

> ***Note:*** This subsection provides a short description of regular expressions. You may skip it if you know what a regex is. In that case go to the task specification right away (see Section 27.1.2).

Imagine you work at a police station that happened to arrest a John Smith who is a suspect in a certain case. In your country the identity of an accused person is to be protected from public, so your job is to obfuscate any mention of him from the press release.

```
function getTxtFromFile(filePath::Str)::Str
    fileTxt::Str = ""
    try
        fileTxt = open(filePath) do file
            read(file, Str)
        end
    catch
        fileTxt = "Can't read '$filePath'. Make sure it exists."
```

[238]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/regex
[239]https://docs.julialang.org/en/v1/stdlib/Pkg/

```
    end
    return fileTxt
end

txt = getTxtFromFile("./loremJohnSmith.txt")
"<<< " * txt[1:200] * " ... >>>"
```

<<< This is a lorem ipsum text from: https://en.wikipedia.org/wiki/Lorem_ipsum it contains a randomly placed name John Smith. It is used for educational purpose only.

Lorem ipsum dolor sit John Smith ame ... >>>

This could be done, e.g. by replacing his last name with its first letter, but it's kind of tedious and boring to do this manually while reading the text. It may be sped up with a word processing program[240]in which `Ctrl+F` is usually a shortcut for a `find` command. In Julia this could be done with eachmatch[241]like so:

```
function getAllMatches(rmi::Base.RegexMatchIterator)::Vec{Str}
    return [regMatch.match for regMatch in rmi]
end

getAllMatches(eachmatch(r"John Smith", txt))[1:2]
```

```
["John Smith", "John Smith"]
```

Here we defined a little function (`getAllMatches`), that will help us to extract the matches as a vector of strings, which is easier to read than the default structure returned by `eachmatch`. Notice, the `r"John Smith"` argument in `eachmatch`. The `r` indicates that the following characters compose no ordinary string, but a special one that is called a regular expression[242](or regex). It may not seem like much right now, but we'll see its potential in a moment.

Once we confirmed the phrase existence we may wish to obfuscate it. Again, in a word processing program this could be done with `Ctrl+H`

[240]https://en.wikipedia.org/wiki/List_of_word_processor_programs
[241]https://docs.julialang.org/en/v1/base/strings/#Base.eachmatch
[242]https://en.wikipedia.org/wiki/Regular_expression

that usually stands for `find and replace` command. In Julia, we would do it with something like:

```
# in Julia strings are immutable
# to make changes permament write it to `txt` and/or to a file
txt = replace(txt, r"John Smith" => "John S")
eachmatch(r"John Smith", txt) |> getAllMatches
```

```
String[]
```

There, we did our job, the identity of an accused person is protected. We may write the file on a disk and send the press report. I imagine now you're wondering what's the big deal with those regexes anyway. For a person with basic computer literacy what we've done doesn't seem particularly advanced. Well, you're right. It it not. That's because in order **to have a regex we need to use some meta-characters, i.e. special symbols that are interpreted beyond their literal meaning. On the other hand, as a general rule, any letter or digit in regex (like `r"John Smith"`) stands for itself**. Overall, the list of meta-characters is rather long, but as stated in the docs[243]it may be found at the PCRE2 syntax manpage[244].

Instead of going through all the meta-characters (admittedly an impossible task for a short book chapter) let me just demonstrate a few of the more important ones with some illustrative examples.

Example 1

```
txt = getTxtFromFile("./loremDates.txt")
"<<< " * txt[1:200] * " ... >>>"
```

<<< This is a lorem ipsum text from: https://en.wikipedia.org/wiki/Lorem_ipsum it contains a randomly placed dates (years). It is used for educational purpose only.

Lorem ipsum dolor sit 2000 amet consec … >>>

[243]https://docs.julialang.org/en/v1/manual/strings/#man-regex-literals
[244]https://www.pcre.org/current/doc/html/pcre2syntax.html

This time, since I study for an exam, my `txt` contains a passage from a history book. I would like to extract the dates from it to make sure I know them all. Let's say that the dates cover years between 1000 AD and the present. Doing a standard string search is no good, after all I would have to check like a thousand numbers. But wait, a simple regex can save me a lot of work. Observe:

```
eachmatch(r"[0-9][0-9][0-9][0-9]", txt) |> getAllMatches
```

```
["2000", "1989", "1517", "1492", "1410", "1918", "1969", "1776", "2001"]
```

This returned all 4-digit sets in the order they appear in the text (left to right, top to bottom).

In the regex (`r"..."`), the `[...]` is a positive character class that matches any of the enclosed characters. Therefore, `[0123456789]` would mean match any character used to represent a digit (`0` or `1` or `2` or …). In general the contents of a positive character class are interpreted literally with the exception of `\`, `^` at the beginning, and `-` between two characters. In the last case, the hyphen (`-`) means any character within a range. Typically its used in the following configurations: `[0-9]`, `[a-z]`, `[A-Z]`, `[A-z]`, or `[A-z0-9]`. The range is likely determined based on the underlying codes (e.g. like ASCII[245]). Therefore, if you want to match any letter (capital or small) the regex must be written as `[A-z]` and not `[a-Z]`. Anyway, in our case a regex of the form `r"[0-9][0-9][0-9][0-9]"`) means match a digit (`[0-9]`) followed by a digit (`[0-9]`), followed by a digit (`[0-9]`), followed by a digit (`[0-9]`) (exactly 4 digits in a row).

Interestingly, we could save ourselves even more typing by using other meta-characters for this problem, i.e.

```
eachmatch(r"[0-9]{4}", txt) |> getAllMatches
```

```
["2000", "1989", "1517", "1492", "1410", "1918", "1969", "1776", "2001"]
```

[245]https://en.wikipedia.org/wiki/ASCII

The `{4}` means exactly 4 repetitions of a preceding character class (which is `[0-9]`, so a digit).

Some newer regex engines allow to shorten it even more:

```
eachmatch(r"\d{4}", txt) |> getAllMatches
```

```
["2000", "1989", "1517", "1492", "1410", "1918", "1969", "1776", "2001"]
```

Where `\d` means any digit (in general `\` gives a special meaning to the following ordinary character) and `{4}` still designates exactly 4 repetitions of a previous token.

Example 2

```
txt = getTxtFromFile("./loremDollarsDates.txt")
"<<< " * txt[1:200] * " ... >>>"
```

<<< This is a lorem ipsum text from: https://en.wikipedia.org/wiki/Lorem_ipsum it contains a randomly placed amounts of money and years. It is used for educational purpose only.

Lorem ipsum dolor sit $11 ... >>>

This time, we got a text that contains both dollars quota (in `$123` format) and dates, but we're interested only in the former. Let's say we want to add them up to find out how much do we need to pay.

First, we'll try to get the numbers out. If we assume for a moment that the amount of money is at least 3 digits long then for our first try we might go with:

```
eachmatch(r"\d.+\d", txt) |> getAllMatches
```

```
[
"112 amet consectetur 1234",
"200",
"173 exercitation ullamco 1492",
"1180 Duis aute irure dolor in $122",
"113 cillum dolore eu $3333",
```

```
"444 sint occaecat cupidatat non $212",
"534"
]
```

Here `\d` means a digit, `.` is any character (except for newline), and `+` stands for one or more of the preceding tokens (so match a digit followed by one or more characters, followed by a digit). There is a small problem though, we caught more than we wanted. That's because by default, regexes are greedy (usually they match as much as they can until a line ends). If we want to make it more temperate we need to follow `.+` with `?` (one or more characters, but as few as you can to fulfill the condition).

```
eachmatch(r"\d.+?\d", txt) |> getAllMatches
```

```
[
"112",
"123",
"200",
"173",
"149",
"118",
"0 Duis aute irure dolor in \$1",
"113",
"333",
"444",
"212",
"534"
]
```

An improvement, but we're still not there. Let's try again.

```
eachmatch(r"\d{1,}", txt) |> getAllMatches
```

```
[
"112"
"1234"
"200"
"173"
"1492"
"1180"
"122"
```

```
"113"
"3333"
"444"
"212"
"534"
]
```

Pretty good, here `{i,j}` means between `i` and `j` (inclusive - inclusive) occurrences of the previous token (`\d`). `{,j}` stands for 0 to `j` and `{i,}` stands for `i` or more. Therefore, we only match 1 or more digits in a row, so it seems that we are finally there. Well, not quite, right now we got no way to tell which digits denote money and which years (they're from the file `loremDollarsDates.txt`).

> ***Note:*** You need to be precise while typing the quantifiers. Typing `eachmatch(r"\d{1, }", txt) |> getAllMatches` (it contains an extra space in `\d{1, }`) will give you no matches (empty vector).

Let's try to change our regex a bit to extract only dollars.

```
eachmatch(r"$\d{1,}", txt) |> getAllMatches
```

```
String[]
```

Hmm, we wanted to extract a dollar symbol `$` with all the following digits. Oddly enough that seemed to have failed. That's because `$` is a meta-character that denotes end of a subject (usually end of a line or end of a string). So, actually what we said with `$\d{1,}` was: find digits after the end of a string. An impossible task, hence the empty vector as a result. If we want `$` to be interpreted as a regular dollar symbol we need to proceed it with `\` (`\` gives a special meaning to an ordinary character, like in `\d`, and strips it away from a special character like `$`).

```
eachmatch(r"\$\d{1,}", txt) |> getAllMatches
```

```
[
"$112",
"$200",
"$173",
"$1180",
"$122",
"$113",
"$3333",
"$444",
"$212",
"$534"
]
```

Finally, we can add it up using, e.g. this few liner:

```
eachmatch(r"\$\d{1,}", txt) |> getAllMatches |>
vecStrDollars -> replace.(vecStrDollars, "\$" => "") |>
vecStrNumbers -> parse.(Int, vecStrNumbers) |>
sum
```

```
6423
```

And voila, we're done. Notice, however, that the regex isn't perfect. For example, it doesn't handle correctly the amounts of money that contain floating point values (or negative quotas). If that were the requirement, we would would have to improve upon it.

Example 3

This time we got a few random telephone numbers.

```
Rnd.seed!(9)
telNums = [join(Rnd.rand(string.(0:9), 9)) for _ in 1:3]
```

```
["304039945", "545946090", "818309467"]
```

Our task is to convert them into a more readable form, e.g. xxx-xxx-xxx.

```
replace.(telNums, r"(\d{3})(\d{3})(\d{3})" => s"\1-\2-\3")
```

```
[
    "304-039-945",
    "545-946-090",
    "818-309-467"
]
```

The new elements here are `()` and `\1`, `\2`, `\3`. Those are capture groups and back-references, respectively. Therefore, `(\d{3})` in a regex (`r""`) means capture any three digits in a row and remember them, whereas `\1` in the substitution (`s""` - denotes a substitution string that may use meta-characters) means: use the first captured and remembered group (by analogy `\2` is for the second captured group and `\3` is for the third).

Example 4

In Section 4 we dealt with two-way text transformations between camelCase and snake_case.

Let's do this with regexes. We'll start with `camelCasedWords`:

```
camelCasedWords = [
    "helloWorld", "niceToMeetYou", "translateToEnglish"
]

eachmatch.(r"([A-Z])", camelCasedWords) .|> getAllMatches
```

```
[
    ["W"],
    ["T", "M", "Y"],
    ["T", "E"]
]
```

First, we capture (`()`) any capital letter (`[A-Z]`) in a string. Now we would like to lowercase it. Per pcre2 syntax manual[246]we should be able to do this using `\l` escape sequence (it means lowercase next character), but for whatever reason the following snippet throws an error:

[246]https://www.pcre.org/current/doc/html/pcre2syntax.html#SEC32

```
replace.(camelCasedWords, r"([A-Z])" => s"\l\1")
```

No, biggie. Julia allows the second argument of `replace` to be a pair[247] of the form `regex => function that operates on a matched string`. We can use that to our advantage:

```
# no need for (), since we don't use backreferences anyway
replace.(camelCasedWords, r"[A-Z]" => lowercase)
```

```
[
"helloworld",
"nicetomeetyou",
"translatetoenglish"
]
```

Almost there, we just need to precede the lower-cased letter with _. This could be done by using an anonymous function, e.g. like this:

```
replace.(camelCasedWords, r"[A-Z]" => AtoZ -> "_" * lowercase(AtoZ))
```

```
[
"hello_world",
"nice_to_meet_you",
"translate_to_english"
]
```

or like that (here we use a template string):

```
replace.(camelCasedWords, r"[A-Z]" => AtoZ -> "_$(lowercase(AtoZ))")
```

```
[
"hello_world",
"nice_to_meet_you",
"translate_to_english"
]
```

Nice.

[247] https://docs.julialang.org/en/v1/base/collections/#Core.Pair

Now, it's time for the opposite transformation.

```
snakeCasedWords = ["hello_world",
    "nice_to_meet_you", "translate_to_english"
]

replace.(snakeCasedWords, r"_[a-z]" => _atoz -> uppercase(_atoz[2:end]))
# or
replace.(snakeCasedWords,
    r"_[a-z]" => _atoz -> uppercase(strip(_atoz, '_')))
```

```
[
"helloWorld",
"niceToMeetYou",
"translateToEnglish",
]
```

Wow, that felt like a breeze.

Overall, the two lines of code (`replace.(camelCasedWord, etc.)` and `replace.(snakeCasedWords, etc.)`) are the equivalent of roughly 20 lines of code in Section 4.2 . And that's how it usually is, regexes are more succinct than the traditional functions, although they're not necessarily faster to write (especially if that is your first encounter with the subject).

Summary

Here's a quick reminder of what we learned about regexes and meta-characters:

1. in general any letter or digit that occurs in regex stands for itself;
2. a positive character class is denoted by square brackets and is often used to capture a range of characters, like `[a-z]`, `[A-Z]`, `[0-9]`, `[A-z]`, or `[A-z0-9]`;
3. `{}` is a quantifier, it specifies a quantity of the previous token, where `{i}`, `{i,j}`, `{i,}`, and `{,j}` mean: exactly `i`, between `i` and `j`, at least `i` and up-to `j` previous tokens, respectively;
4. `\` bestows a special meaning on an ordinary character (`\d` denotes any digit), or strips it away from a special character (`$` - is end of a string, wheres `\$` is a dollar symbol);

5. `(sth)` inside of `r""` stands for capture and remember, whereas `\1` in `s""` denotes back-reference to the first capture;
6. the second argument of replace is a pair[248]of the form: `r"" => ""` (regex => regular string), `r"" => s""` (regex => substitution string), `r"" => function` (regex => function that accepts a string and returns a string possibly applying some transformations on the way).

Regex Tasks

OK, time to put what you've learned to good use. If, while solving the tasks, you need a visual assistant that helps you with regular expressions, then you may try e.g. regex101[249].

Regex Task 1

You got a series of dates in the US format "MM.DD.YYYY":

```
datesMMDDYYYY = ["01.04.2025", "11.01.2018", "12.31.1999", "03.20.2026"]
```

The format is confusing to some (e.g. European) people. Change it to a less ambiguous "YYYY-MM-DD" configuration.

Regex Task 2

Read the contents of `loremMail.txt` that is to be found in the code snippets[250]. It contains random e-mail addresses (with repetitions). Use Julia to list the unique e-mail addresses found in the text.

Regex Task 3

Here's a vector of random names:

```
# random names
names = ["Mary Johnson", "Eve Smith", "Tom Brown"]
```

Swap the names order with a regex ("Adam Smith" should become "Smith, Adam") and sort them alphabetically in ascending order.

---

[248]https://docs.julialang.org/en/v1/base/collections/#Core.Pair
[249]https://regex101.com/
[250]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/regex

Can you do the same, but while accounting for possible middle names.

```
# random names
names = ["Jane Johnson", "Mary Jane Doe", "Peter Smith", "Adam Tom
Brown"]
```

To add a small tweak, I want you to swap the names, abbreviate the middle name ("John Daniel Smith" should become "Smith, John D.", whereas "Adam Smith" should become "Smith, Adam") and then sort them alphabetically in ascending order.

Regex Task 4

In Section 32.2 we wrote a `fmt` function to format numbers to something like: "123,456 USD".

Write a program (possibly a regex or regexes + some extra code) that will convert those numbers to the desired form (place `,` after every three numbers from right).

```
nums = [0, 1, 12, 123, 1234, 12345,
    123456, 1234567, 12345678, 123456789]
```

Can you modify your program so that it handles the following numbers correctly as well (e.g. `12345.678` should become "12,345.68 USD"):

```
nums = [0, 0.1, 1, 1.2, 12., 12.34, 123.456,
    1234, 12345, 12345.67, 123456.7, 1234567.89]
```

Well, let's find out. Good luck.

Solution

Regex Solution 1

The solution is pretty straightforward if you read through Example 1 and 3 in Section 27.1.1 .

```
replace.(datesMMDDYYYY, r"(\d{2})\.(\d{2})\.(\d{4})" => s"\3-\1-\2")
```

```
[
"2025-01-04",
"2018-11-01",
"1999-12-31",
"2026-03-20"
]
```

We just go and capture the months (the first pair of digits, `(\d{2})`, followed by a literal dot, `\.`), the days (second pair of digits, `(\d{2})`, followed by a literal dot, `\.`) and the years (last four digits, `(\d{4})`). In the substitution string we reference them back in the appropriate order (`\3`, `\1`, and `\2`) separated by hyphens (`-`). So, we replaced the whole match (`(\d{2})\.(\d{2})\.(\d{4})`) with the remembered digits in the right order and with the right separators (`-`). And that's it. Finito.

Regex Solution 2

```
txt = getTxtFromFile("./loremMail.txt")
"<<< " * txt[1:200] * " ... >>>"
```

<<< This is a lorem ipsum text from: https://en.wikipedia.org/wiki/Lorem_ipsum it contains a randomly placed fake e-mail addresses (kid of). It is used for educational purpose only.

Lorem ipsum dolor sit … >>>

Surprisingly, it seems that a proper regex for e-mail validation is pretty complex (see here[251]). Still, we can go a far way with a much simpler one, which in our particular case should do the trick:

```
eachmatch(r"[A-z0-9._\-]+@[A-z0-9._\-]+", txt) |> getAllMatches |>
unique
```

```
[
    "tom@write2me.com",
    "potential_contact@hello.pl",
    "another.contact@yyy.es",
```

[251]https://stackoverflow.com/questions/201323/how-can-i-validate-an-email-address-using-a-regular-expression

```
    "other-potential-contact@hello.pl",
    "eve@write2me.com",
    "eve2@write2me.com"
]
```

The regex is composed of a few parts, but mostly of `[A-z0-9._\-]`. It searches for:

1) any letter (`A-z`, an email may contain a capital letter, although in general, they are case-insensitive[252]), or
2) a digit (`0-9`), or
3) a literal dot (`.` inside a positive character class is just a dot, although in general inside a regex it stands for any character except for newline), or
4) an underscore (_), or
5) a literal hyphen (`\-`, likely we didn't have to precede it with `\` since it wasn't between other 2 characters).

This positive character class must be repeated at least one time (+) before the `@`, symbol. On the other hand, the `@` symbol must be followed by at least one (+) character class that we already discussed (`[A-z0-9._\-]`). Notice, that there is no need to add `?`, after the `+` to make a non-greedy match. That is because the email addresses, are separated by one or more spaces and the positive character class, (`[A-z0-9._\-]`) does not include spaces.

Regex Solution 3

Swapping the names is a piece of cake. We just use capture groups (`(...)`) that contain one or more (+) letters (`[A-z]`) per word and are separated by a white-space character. In the substitution string (`s""`) we use back-references in reversed order (`\2` and `\1`) and separate them with a comma (`","`):

```
# random names
names = ["Mary Johnson", "Eve Smith", "Tom Brown"]

replace.(names, r"([A-z]+) ([A-z]+)" => s"\2, \1")
```

[252] https://en.wikipedia.org/wiki/Case_sensitivity

```
[
    "Johnson, Mary",
    "Smith, Eve",
    "Brown, Tom"
]
```

Once we got them formatted sorting shouldn't be a problem either:

```
replace.(names, r"([A-z]+) ([A-z]+)" => s"\2, \1") |> sort
```

```
[
    "Brown, Tom",
    "Johnson, Mary",
    "Smith, Eve"
]
```

OK, time for some more complicated names:

```
# random names
names = [
    "Jane Johnson",
    "Mary Jane Doe",
    "Peter Smith",
    "Adam Tom Brown"
]
```

Let's build our regex step by step. We start by matching a middle name (if there is one).

```
eachmatch.(r" [A-z]+ ", names) .|> getAllMatches
```

```
[
    [],
    [" Jane "],
    [],
    [" Tom "]
]
```

Here we search for a word between two spaces, more specifically: a white-space character, at least one letter (`[A-z]+`) and a white-space character.

Time to abbreviate the middle name:

```
replace.(names, r" ([A-Z])[a-z]+ " => s" \1. ")
```

```
[
    "Jane Johnson",
    "Mary J. Doe",
    "Peter Smith",
    "Adam T. Brown"
]
```

For that we modified the previous regex. This time we looked for a white-space character, one capital letter (`[A-Z]`), at least one small letter (`[a-z]+`) and a white-space character. Out of the whole match (`" ([A-Z])[a-z]+ "`) we captured (`()`) and remembered only the capital letter, which we used in the substitution string (`s""`) followed by a literal dot (`\1.`). Therefore, we replaced the whole match (`" ([A-Z])[a-z]+ "`) by its first capture group that we memorized and referred back to with (`\1`).

Now, time for the swap:

```
replace.(names, r" ([A-Z])[a-z]+ " => s" \1. ") |>
abbrevNames -> replace.(abbrevNames, r"([A-z .]+) ([A-z]+)" => s"\2,
\1")
```

```
[
    "Johnson, Jane",
    "Doe, Mary J.",
    "Smith, Peter",
    "Brown, Adam T."
]
```

Here, instead of being clever and building a one complicated regex, we just passed the result of one `replace` function as an input to another `replace` function. The second regex looks for at least one letter, space or literal dot (`[A-z .]+`, this captures as many consecutive words as it can because of the greediness) followed by one word (`[A-z]+`, one or more letters). We captured the words with `()` and swapped them with back-references (`\2` and `\1`), while putting a comma (`,`) between them.

OK, now for the last step, sorting:

```
replace.(names, r" ([A-Z])[a-z]+ " => s" \1. ") |>
abbrevNames -> replace.(abbrevNames, r"([A-z .]+) ([A-z]+)" => s"\2,
\1") |>
sort
```

```
[
    "Brown, Adam T.",
    "Doe, Mary J.",
    "Johnson, Jane",
    "Smith, Peter"
]
```

And we're done.

Regex Solution 4

OK, time for a tough challenge. Let's properly format `nums` using only the regex techniques we learned so far (see Section 27.1.1) + some built-in Julia functions. My first try would look something like:

```
nums = [0, 1, 12, 123, 1234, 12345,
    123456, 1234567, 12345678, 123456789]

replace.(string.(nums), r"(\d{3})" => s"\1,")
```

```
# no commas separating elts of vector, to make it more legible
[
    "0"
    "1"
    "12"
    "123,"
    "123,4"
    "123,45"
    "123,456,"
    "123,456,7"
    "123,456,78"
    "123,456,789,"
]
```

Overall, we did a pretty good job. First, we changed the integers (`nums`) into strings (by using `string` function). Next, we said: while moving left to right (default direction for a regex engine) match exactly three

digits (`\d{3}`) and remember them (`(...)`). Finally, insert the remembered digits followed by a comma (`"\1,"`). There is a small problem though. The triplets are matched starting from left side instead of the right (which we would prefer). Not a problem, we'll just reverse the string before transformation (putting commas).

```
replace.(reverse.(string.(nums)), r"(\d{3})" => s"\1,")
```

```
# no commas separating elts of vector, to make it more legible
[
    "0"
    "1"
    "21"
    "321,"
    "432,1"
    "543,21"
    "654,321,"
    "765,432,1"
    "876,543,21"
    "987,654,321,"
]
```

OK, the commas are placed every three digits from right (if you consider the original numbers). Now we would like to remove the stray comma at the end of some lines (`r",$" => ""`) and reverse the string again (to restore the original order):

```
replace.(reverse.(string.(nums)), r"(\d{3})" => s"\1,") |>
reversedNums -> replace.(reversedNums, r",$" => "") .|> reverse
```

```
# no commas separating elts of vector, to make it more legible
[
    "0"
    "1"
    "12"
    "123"
    "1,234"
    "12,345"
    "123,456"
    "1,234,567"
    "12,345,678"
    "123,456,789"
]
```

To make it slightly more elegant we can enclose the entire procedure into a function:

```
function fmtMoney(n::Int)::Str
    @assert n >= 0 "n must be >= 0"
    result::Str = replace(reverse(string(n)), r"(\d{3})" => s"\1,")
    return replace(result, r",$" => "") |> reverse
end
```

And use it for money formatting:

```
fmtMoney.(nums) .* " USD"
```

```
[
    "0 USD",
    "1 USD",
    "12 USD",
    "123 USD",
    "1,234 USD",
    "12,345 USD",
    "123,456 USD",
    "1,234,567 USD",
    "12,345,678 USD",
    "123,456,789 USD"
]
```

The above `fmtMoney` is a five line regex equivalent of the fifteen lines long `getFormattedMoney` from Section 31.1.4 . Likely, it could be shortened even more by applying lookahead assertions[253](we didn't use them, since they had not been discussed in Section 27.1.1).

Now, let's go one step further and try to format also decimals. For that, we'll split a number into dollars (integers) and cents (two digits after comma, rounded if necessary).

```
function getDollarsCents(money::Flt)::Tuple{Int, Int}
    @assert money >= 0 "money must be >= 0"
    integralPart::Int = floor(Int, money)
    decimalPart::Flt = money % 1
    return (integralPart, round(Int, decimalPart*100))
end
```

[253]https://en.wikipedia.org/wiki/Regular_expression#Assertions

Once we got it, we will use `fmtMoney(n::Int)::Str` to format the dollars to which we'll append the cents:

```
function fmtMoney(n::Flt)::Str
    @assert n >= 0 "n must be >= 0"
    dollars::Int, cents::Int = getDollarsCents(n)
    result::Str = fmtMoney(dollars)
    return string(result, ".", cents)
end
```

Time to test it out:

```
nums = [0, 0.1, 1, 1.2, 12., 12.34, 123.456,
    1234, 12345, 12345.67, 123456.7, 1234567.89]

fmtMoney.(nums) .* " USD"
```

```
[
    "0.0 USD",
    "0.10 USD",
    "1.0 USD",
    "1.20 USD",
    "12.0 USD",
    "12.34 USD",
    "123.46 USD",
    "1,234.0 USD",
    "12,345.0 USD",
    "12,345.67 USD",
    "123,456.70 USD",
    "1,234,567.89 USD"
]
```

Looks like we finished the job. Regular expressions are worth learning even at a relatively basic level. Sometimes they can really speed things up or reduce the amount of code.

Molar Mass

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[254](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[255].

Problem

I remember that when I was in high school we used to have a lot of chemistry classes filled with problem solving. The key part of them was to calculate molar masses of different molecules. That segment, although essential, was rather boring. So I always wanted to have something that would speed up the calculations for me.

This time your job is to write a solver that will calculate a molar mass of a chemical formula. Make sure your solution works by using it on the following input:

- CH_4 (methane, natural gas, fossil fuel),
- H_2O (water),
- HCl (hydrochloric acid produced in your stomach),
- CO_2 (carbon dioxide, breath it out),
- C_3H_8 (propane, natural gas, fossil fuel),
- C_2H_5OH (ethanol, is it time to quit drinking?),
- $(CH_3)_2CO$ (acetone, nail polish remover),
- $NaCl$ (sodium chloride, kitchen salt),
- CH_3COOH (acetic acid, it's in vinegar),
- H_2CO_3 (carbonic acid present in your blood),
- $C_6H_{12}O_6$ (glucose, it gives you energy),

[254]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/molar_mass

[255]https://docs.julialang.org/en/v1/stdlib/Pkg/

- $C_{11}H_{12}N_2O_2$ (tryptophan, amino acid, builds proteins),
- $CH_3(CH_2)_{14}COOH$ (palmitic acid, builds fat),
- $C_{34}H_{32}O_4N_4Fe$ (heme B, located in your red blood cells)
- $Ca_{10}(PO_4)_6(OH)_2$ (calcium hydroxyapatite found in your bones)
- $C_{169719}H_{270466}N_{45688}O_{52238}S_{911}$ (titin, a protein that builds your muscles)

Below I paste the formulas and their masses for testing purposes.

```
formulas = ["CH4", "H2O", "HCl", "CO2", "C3H8", "C2H5OH", "(CH3)2CO",
            "NaCl", "CH3COOH", "H2CO3", "C6H12O6", "C11H12N2O2",
            "CH3(CH2)14COOH", "C34H32O4N4Fe", "Ca10(PO4)6(OH)2",
            "C169719H270466N45688O52238S911"]
masses = [16.043, 18.01528, 36.46, 44.01, 44.097, 46.069, 58.08, 58.443,
          60.052, 62.03, 180.156, 204.229, 256.43, 616.487, 1004.61,
          3_816_030]
```

A list of chemical elements and their masses is to be found, e.g. on this Wikipedia page[256].

For simplicity, you may assume, that a chemical formula (at least the one at a high school level) is composed of ASCII[257] characters (capital/small letters + digits) and non-nested brackets.

Solution

Let's start by defining the elements mass table.

```
const ELTS_MASS_TBL = Dict{Str, Flt}(
    "H" => 1.008, "He" => 4.0026, "Li" => 6.94, "Be" => 9.0122,
    "B" => 10.81, "C" => 12.011, "N" => 14.007, "O" => 15.999,
    "F" => 18.998, "Ne" => 20.18, "Na" => 22.99, "Mg" => 24.305,
    "Al" => 26.982, "Si" => 28.085, "P" => 30.974, "S" => 32.06,
    "Cl" => 35.45, "Ar" => 39.95, "K" => 39.098, "Ca" => 40.078,
    "Sc" => 44.956, "Ti" => 47.867, "V" => 50.942, "Cr" => 51.996,
    "Mn" => 54.938, "Fe" => 55.845, "Co" => 58.933, "Ni" => 58.693,
    "Cu" => 63.546, "Zn" => 65.38, "Ga" => 69.723, "Ge" => 72.63,
    "As" => 74.922, "Se" => 78.971, "Br" => 79.904, "Kr" => 83.798,
    "Rb" => 85.468, "Sr" => 87.62, "Y" => 88.906, "Zr" => 91.224,
    "Nb" => 92.906, "Mo" => 95.95, "Tc" => 96.906, "Ru" => 101.07,
    "Rh" => 102.91, "Pd" => 106.42, "Ag" => 107.87, "Cd" => 112.41,
```

[256] https://en.wikipedia.org/wiki/List_of_chemical_elements#List
[257] https://en.wikipedia.org/wiki/ASCII

```
    "In" => 114.82, "Sn" => 118.71, "Sb" => 121.76, "Te" => 127.6,
    "I" => 126.9, "Xe" => 131.29, "Cs" => 132.91, "Ba" => 137.33,
    "La" => 138.91, "Ce" => 140.12, "Pr" => 140.91, "Nd" => 144.24,
    "Pm" => 144.913, "Sm" => 150.36, "Eu" => 151.96, "Gd" => 157.25,
    "Tb" => 158.93, "Dy" => 162.5, "Ho" => 164.93, "Er" => 167.26,
    "Tm" => 168.93, "Yb" => 173.05, "Lu" => 174.97, "Hf" => 178.49,
    "Ta" => 180.95, "W" => 183.84, "Re" => 186.21, "Os" => 190.23,
    "Ir" => 192.22, "Pt" => 195.08, "Au" => 196.97, "Hg" => 200.59,
    "Tl" => 204.38, "Pb" => 207.2, "Bi" => 208.98, "Po" => 208.982,
    "At" => 209.987, "Rn" => 222.018, "Fr" => 223.02, "Ra" => 226.025,
    "Ac" => 227.028, "Th" => 232.04, "Pa" => 231.04, "U" => 238.03,
    "Np" => 237.048, "Pu" => 244.064, "Am" => 243.061, "Cm" => 247.070,
    "Bk" => 247.070, "Cf" => 251.08, "Es" => 252.083, "Fm" => 257.095,
    "Md" => 258.098, "No" => 259.101, "Lr" => 266.12, "Rf" => 267.122,
    "Db" => 268.126, "Sg" => 269.128, "Bh" => 270.133, "Hs" => 269.134,
    "Mt" => 277.154, "Ds" => 282.166, "Rg" => 282.169, "Cn" => 286.179,
    "Nh" => 286.182, "Fl" => 290.192, "Mc" => 290.196, "Lv" => 293.205,
    "Ts" => 294.211, "Og" => 295.216
)
```

Note. Using `const` with mutable containers like vectors or dictionaries allows to change their contents later on, e.g., with `push!`. So the `const` used here is more like a convention, a signal that we do not plan to change the containers in the future. If we really wanted an immutable container then we should consider a(n) (immutable) tuple. Anyway, some programming languages suggest that `const` names should be declared using all uppercase characters to make them stand out. Here, I follow this convention.

All right, now we may write a simple formula solver, but first some helper functions.

```
const MASS_FALLBACK = typemin(Flt)

function getEltMass(elt::Str)::Flt
    return isempty(elt) ? 0.0 : get(ELTS_MASS_TBL, elt, MASS_FALLBACK)
end

# 1 is neutral for multiplication
function str2int(s::Str, def::Int=1)::Int
    try
        return parse(Int, s)
    catch
```

```
        return def
    end
end
```

First we define `getEltMass` that for an empty string (`isempty(elt)` - no element at all) returns the mass equal `0.0` (no mass at all). Otherwise, it fetches the mass out of `ELTS_MASS_TBL`. If the element (`elt`) is not there it returns `MASS_FALLBACK` which is `typemin(Flt)`. That last expression is equal to `-Inf`, a special value that we want to use as an indicator that something went wrong. In general, `-Inf` propagates since almost any value added to `-Inf` is `-Inf`.

To calculate the molar mass of a molecule we plan to proceed atom by atom. At times an element will be followed by a number (by which we will multiply its mass). Therefore, we define `str2int` that `try`ies to transform a string (`s`) into an integer (`parse(Int, s)`). Normally, when the `parser` fails (e.g. in the case of an empty string) it throws an error. But we catch[258]it and `return` a default value (`def`) instead (here we go with 1 as it's neutral for multiplication).

OK, now back to the simple formula solver.

```
function getMolMassSimple(formula::Str)::Flt
    mass::Flt = 0.0
    curElt::Str = ""
    curNum::Str = ""
    for c in formula # c - a character in formula
        if isuppercase(c)
            mass += getEltMass(curElt) * str2int(curNum)
            curElt = string(c)
            curNum = ""
        elseif islowercase(c)
            curElt *= c
        elseif isdigit(c)
            curNum *= c
        else # should not happen
            return MASS_FALLBACK
        end
    end
    mass += getEltMass(curElt) * str2int(curNum)
```

[258]https://docs.julialang.org/en/v1/manual/control-flow/#The-try/catch-statement

```
    return mass
end
```

The algorithm is rather mundane. We start by initializing a few variables, `mass` - to hold the result, `curElt` to keep the currently examined element, `curNum` - that stores current number of atoms of a given element. Next, we traverse the `formula` one character at a time (`for c in formula`). If the examined character is a capital letter (`isuppercase(c)`) we calculate the mass of previously stored element (`curElt` multiplied by `curNum`) and add it to `mass` (`mass += etc.`). Of course, we remember to reset the `curElt` and `curNum` to their new values. If a character is a small letter (`elseif islowercase(c)`) or a digit (`elseif isdigit(c)`) we just append it to the previously encountered element (`curElt *= c`) or number (`curNum *= c`), respectively. If the character (`c`) passes through all our guards (`else`) then we return `MASS_FALLBACK`. Anyway, after leaving the `for` loop and before the `return` statement we must do some cleanup. That's because the mass is calculated only when we encounter a capital letter (`isuppercase(c)`), so we need to remember to add a mass of the final element (`mass += etc.`) in a formula before we `return` from our function.

Let's do some minimal testing.

```
isSameMass(x::Flt, y::Flt)::Bool = isapprox(x, y, rtol=0.0001)

map(isSameMass, getMolMassSimple.(formulas[1:6]), masses[1:6])
```

```
Bool[1, 1, 1, 1, 1, 1]
```

For that we defined `isSameMass` using single expression function syntax[259]. Inside it relies on isapprox[260]with relative tolerance (`rtol`) set to `0.0001`. The function is used to account for any rounding errors in `ELTS_MASS_TBL` or `masses`. It considers two numbers equal if they

[259]https://en.wikibooks.org/wiki/Introducing_Julia/Functions#Single_expression_functions

[260]https://docs.julialang.org/en/v1/base/math/#Base.isapprox

differ by no more than 1/10,000th part (i.e. `isSameMass(10_000.0, 9_999.0)` is `true`, but `isSameMass(10_000.0, 9_998.9)` is `false`). Anyway, all the masses of all the tested simple formulas were roughly equal to those in the `masses` vector (`1` is an abbreviated printout for `true`, `0` would be an abbreviated printout for `false`).

OK, time for more complicated formulas.

```
function isInSimpleChemFormula(c::Char)::Bool
    return isuppercase(c) || islowercase(c) || isdigit(c)
end

function getMolMass(formula::Str)::Flt
    curGroup::Str = ""
    curMultiplier::Str = ""
    bracketEnded::Bool = false
    groups::Vec{Str} = []
    multipliers::Vec{Int} = []
    for c in formula
        if isInSimpleChemFormula(c) && !bracketEnded
            curGroup *= c
        elseif isdigit(c) && bracketEnded
            curMultiplier *= c
        elseif isuppercase(c) && bracketEnded
            bracketEnded = false
            push!(groups, curGroup)
            push!(multipliers, str2int(curMultiplier))
            curGroup = string(c)
            curMultiplier = ""
        elseif c == '('
            push!(groups, curGroup)
            push!(multipliers, str2int(curMultiplier))
            curGroup = ""
            curMultiplier = ""
        elseif c == ')'
            bracketEnded = true
        else # should never happen
            return MASS_FALLBACK
        end
    end
    push!(groups, curGroup)
    push!(multipliers, str2int(curMultiplier))
    return sum(getMolMassSimple.(groups) .* multipliers)
end
```

```
getMolMass (generic function with 1 method)
```

Here, we decided to split a complicated `formula` into `groups` of simple formulas that we already can solve correctly. We also added `multipliers` for our `groups` and `bracketEnded`, a flag that tells us whether we just examined the end of a parenthesized group (`c == ')'`). Just like in `getMolMassSimple` we move one character at a time (`for c in formula`) and do some checks. If a character belongs to a simple formula (`if isInSimpleChemFormula(c)`) and (&&) it is not right after the parentheses (`!bracketEnded`) then we just append it to the current group (`curGroup *= c`). If it is a digit (`elseif isdigit(c)`) right after the bracket end (`&& bracketEnded`) we append it to the multiplier for the current group (`curMultiplier`). Else, if it is a capital letter (`elseif isuppercase(c)`) that follows the closing bracket (`&& bracketEnded`) then we reset `bracketEnded` and `push` previous group and multiplier to the appropriate collections. Additionally, we do some cleanup (reset `curGroup` and `curMultiplier`). Likewise, if the parentheses just started (`elseif c == '('`) we push the previous group and multiplier to the collections and reset the current values (`curGroup = ""` and `curMultiplier = ""`). Of course we must remember to set the `bracketEnded` flag to `true` when the parentheses end (`elseif c == ')'`). Once again, before we `return` our result we `push` the last group and multiplier (cleanup). Our final result is just a `sum` of masses of all groups (`getMolMassSimple.(groups)`) multiplied by the appropriate numbers (`.* multipliers`).

Let's see how we did.

```
map(isSameMass, getMolMass.(formulas), masses)
```

```
Bool[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Apparently, we did just fine.

The solution works, but may be considered inelegant and hard to follow (long functions, mostly imperative programming style[261]).

[261]https://en.wikipedia.org/wiki/Imperative_programming

Let's try to change it using what we learned about regexes in Section 27 .

Again, we'll start with simple formulas, but first some helper functions.

```
function getPatternsInTxt(pattern::Regex, txt::Str)::Vec{Str}
    return [regMatch.match for regMatch in eachmatch(pattern, txt)]
end
```

`getPatternsInTxt` is just a modification and contraction of `eachmatch(etc.) |> getAllMatches` functionality from Section 27.1.1 . It returns the matches as a vector (possibly empty) of strings. We'll use it to extract atoms and their numbers from a simple formula.

```
function getAtomsAndNumbers(simpleFormula::Str)::Vec{Str}
    return getPatternsInTxt(r"[A-Z][a-z]{0,}[0-9]{0,}", simpleFormula)
end

function getAtom(atomAndNumber::Str)::Str
    return getPatternsInTxt(r"[A-Z][a-z]{0,}", atomAndNumber)[1]
end

function getNumberAtEnd(txt::Str)::Str
    nAtoms::Vec{Str} = getPatternsInTxt(r"[0-9]{1,}$", txt)
    return isempty(nAtoms) ? "" : nAtoms[1]
end
```

Here, the functions do what their names promise. While the regexes say:

- `[A-Z][a-z]{0,}[0-9]{0,}` - match exactly one capital letter, followed by none or more small letters, followed by zero or more digits.
- `[A-Z][a-z]{0,}` - match exactly one capital letter, followed by none or more small letters
- `[0-9]{1,}$` - match one or more digits that are at the end of the subject (here a string)

Let's see how they work:

```
formulas[6],
formulas[6] |> getAtomsAndNumbers,
formulas[6] |> getAtomsAndNumbers .|> getAtom,
formulas[6] |> getAtomsAndNumbers .|> getNumberAtEnd
```

```
(
"C2H5OH",
["C2", "H5", "O", "H"],
["C", "H", "O", "H"],
["2", "5", "", ""]
)
```

and

```
formulas[3],
formulas[3] |> getAtomsAndNumbers,
formulas[3] |> getAtomsAndNumbers .|> getAtom,
formulas[3] |> getAtomsAndNumbers .|> getNumberAtEnd
```

```
(
"HCl",
["H", "Cl"],
["H", "Cl"],
["", ""]
)
```

Looks good. We aren't worried about the empty strings in numbers of atoms, since `str2int` will handle them and return `1's`.

OK, time for another simple formula solver.

```
function getmolmasssimple(formula::Str)::Flt
    atomsAndNumbers::Vec{Str} = getAtomsAndNumbers(formula)
    atoms::Vec{Str} = getAtom.(atomsAndNumbers)
    numbers::Vec{Int} = getNumberAtEnd.(atomsAndNumbers) .|> str2int
    atomsMasses::Vec{Flt} = getEltMass.(atoms)
    return sum(atomsMasses .* numbers)
end
```

Here, we used an all lowercase name, since eventually we would like to test the performance of our solvers and we don't want to override `getMolMassSimple`. Anyway, we proceed in a series of a few logical steps (notice the `.` symbols that indicate when a function is used on a

vector). First we subtract atoms and their numbers (`atomsAndNumbers`). Then, we use them (`atomsAndNumbers`) to subtract atoms (`atoms`) and the number of their occurrences (`numbers`). Next, we calculate the masses of our atoms (`atomsMasses`), which we multiply (`.*`) by the numbers of their occurrences (`numbers`) and `sum` it all together.

Time for a test ride.

```
map(isSameMass, getmolmasssimple.(formulas[1:6]), masses[1:6])
```

```
Bool[1, 1, 1, 1, 1, 1]
```

The ride was satisfactory.

Now, before we go to more complicated formulas we'll write some helper functions.

```
function getBracketedGroups(formula::Str)::Vec{Str}
    return getPatternsInTxt(r"\(.+?\)\d{0,}", formula)
end

function getInsideOfBrackets(group::Str)::Str
    return replace(group, "(" => "", r"\)\d{0,}" => "")
end
```

The regex in `getBracketedGroups` is `\(.+?\)\d{0,}` which states:

Match any character (`.`) repeated one or more times (`+`), but as few as possible (`?`) that is inside the literal brackets (`\(` and `\)`). The closing bracket is followed by zero or more digits (`\d{0,}`). Notice that inside a regex `(sth)` - is a capture and remember command (usually used with replace[262]), so we strip the special meaning by using `\` before `(` and `)`.

As for `getInsideOfBrackets` we just `replace` the opening brackets with nothing (`"(" => ""`), and closing bracket followed by optional digits (`r"\)\d{0,}"`) with nothing (`""` - empty string). This effectively removes brackets and their outer surroundings. BTW, did you notice

[262] https://docs.julialang.org/en/v1/base/collections/#Base.replace-Tuple%7BAny,%20Vararg%7BPair%7D%7D

the difference in “`(`“ and “`\)`” when used in normal string and in the regex?

Let’s see how we can use it.

```
formulas[15],
formulas[15] |> getBracketedGroups,
formulas[15] |> getBracketedGroups .|> getInsideOfBrackets,
formulas[15] |> getBracketedGroups .|> getNumberAtEnd
```

```
(
"Ca10(PO4)6(OH)2",
["(PO4)6", "(OH)2"],
["PO4", "OH"],
["6", "2"]
)
```

And again (`String[]` in the output is an empty vector of strings).

```
formulas[6],
formulas[6] |> getBracketedGroups,
formulas[6] |> getBracketedGroups .|> getInsideOfBrackets,
formulas[6] |> getBracketedGroups .|> getNumberAtEnd
```

```
(
"C2H5OH",
String[],
String[],
String[]
)
```

Now, for the ‘full’ solver.

```
function getPair(fst::Str, snd::Str="")::Pair{Str, Str}
    return Pair(fst, snd) # alternative to: return fst => snd
end

function remAll(txt::Str, extras::Vec{Str})::Str
    return replace(txt, map(getPair, extras)...)
end

function getmolmass(formula::Str)::Flt
    groupFormulas::Vec{Str} = getBracketedGroups(formula)
    groupsInsides::Vec{Str} = getInsideOfBrackets.(groupFormulas)
```

```
    groupsMultipliers::Vec{Int} = getNumberAtEnd.(groupFormulas) .|>
str2int
    formula = remAll(formula, groupFormulas)
    grInsMasses::Vec{Flt} = map(getMolMassSimple, groupsInsides)
    return sum(grInsMasses .* groupsMultipliers) +
getmolmasssimple(formula)
end
```

Our plan is to subtract the groups with parentheses (if any) from a `formula` and then to remove them from it (so that only simple part remains). The removal will be done wit the `replace(txt, to_find_in_txt_1 => to_replace_in_txt_1, to_find_in_txt_2 => to_replace_in_txt_2, ...)` syntax. Hence a pair creator (`getPair`) and the remover (`remAll` where `...` unpacks the vector of pairs produced by `map`).

Inside `getmolmass` we:

1) extract the groups with brackets (`groupFormulas`)
2) extract the groups insides (`groupsInsides`)
3) get the multipliers for each group (`groupsMultipliers`)
4) remove the groups from `formula`, so only simple part remains
5) calculate the masses for groups insides (`grInsMasses`)
6) multiply (`.*`) `grInsMasses` by `groupMultipliers` and `sum` the products
7) add the mass (`getmolmasssimple(formula)`) of the remaining simple part to the result

Notice, that strings are immutable so `remAll` does not actually change the `formula` sent as an argument to `getmolmass` (in `formula = remAll(formula, groupFormulas)`).

Time for testing.

```
map(isSameMass, getmolmass.(formulas), masses)
```

```
Bool[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Appears to be working as intended.

We may compare our functions (`getMolMass` and `getmolmass`) with @time[263]macro (just make sure that both the functions are run at least once beforehand, since functions are compiled at their first usage).

```
@time map(isSameMass, getMolMass.(formulas), masses)
# and
@time map(isSameMass, getmolmass.(formulas), masses)
```

Which will return (except for the results) a printout similar to:

```
0.001191 seconds (451 allocations: 14.984 KiB)
# and
0.000589 seconds (1.37 k allocations: 86.281 KiB)
```

The above indicates that our regex version is faster than its counterpart, but it uses up more memory. So as you can see there are always some trade-offs.

Anyway, for more serious benchmarking we should probably use BenchmarkTools.jl[264]as indicated in the documentation. We will demonstrate that briefly in Section 36.2 , but for now you may take a rest.

[263]https://docs.julialang.org/en/v1/base/base/#Base.@time
[264]https://github.com/juliaci/benchmarktools.jl

Tic-Tac-Toe

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[265](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[266].

Problem

Write a program that will enable you to play a simple tic-tac-toe[267] game. It may look like the one in Figure 13 .

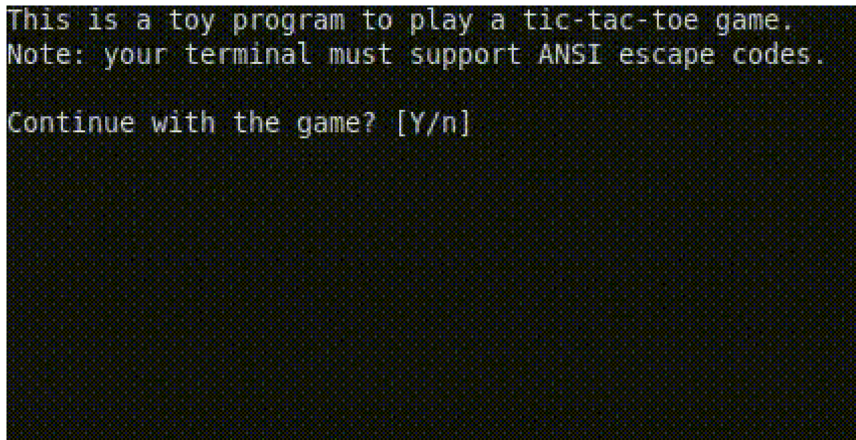


Figure 13: A simple terminal based tic tac toe game (animation works only in an HTML document).

In case you wanted to use the same colors as in the gif above, you may

[265] https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/tic_tac_toe

[266] https://docs.julialang.org/en/v1/stdlib/Pkg/

[267] https://en.wikipedia.org/wiki/Tic-tac-toe

find the ANSI escape codes[268]useful.

Solution

The first decision we must make is the internal representation of our game board. Two candidate data types come to mind right away, a vector or a matrix. Here, I'll go with the first option.

```
function getNewGameBoard()::Vec{Str}
    return string.(1:9)
end
```

Next, we'll define a few constants that will be helpful later on.

```
const PLAYERS = ["X", "O"]
# LINES[1:3] - rows, LINES[4:6] - columns, LINES[7:8] - diagonals
const LINES= [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
    [1, 4, 7],
    [2, 5, 8],
    [3, 6, 9],
    [1, 5, 9],
    [3, 5, 7]
]
```

> Note. Using `const` with mutable containers like vectors or dictionaries allows to change their contents later on, e.g., with `push!`. So the `const` used here is more like a convention, a signal that we do not plan to change the containers in the future. If we really wanted an immutable container then we should consider a(n) (immutable) tuple. Anyway, some programming languages suggest that `const` names should be declared using all uppercase characters to make them stand out. Here, I follow this convention.

The two are: `PLAYERS`, a vector with marks used by each of the players (`"X"` - human, `"O"` - computer) and the coordinates of `LINES` in our game board that we need to check to see if a player won the game. You

[268]https://en.wikipedia.org/wiki/ANSI_escape_code#Colors

could probably be more clever and use enums[269]for the players and list comprehensions for our lines (e.g., `[collect(i:(i+2)) for i in [1, 4, 7]]` to get the rows), but for such a simple case it might be overkill.

OK, time to format the board.

```
# https://en.wikipedia.org/wiki/ANSI_escape_code#Colors
function getGray(s::Str)::Str
    # "\x1b[90m" sets forground color to gray
    # "\x1b[0m" resets forground color to default value
    return "\x1b[90m" * s * "\x1b[0m"
end

function isFree2Take(field::Str)::Bool
    return !(field in PLAYERS)
end

function colorFieldNumbers(board::Vec{Str})::Vec{Str}
    result::Vec{Str} = copy(board)
    for i in eachindex(board)
        if isFree2Take(board[i])
            result[i] = getGray(board[i])
        end
    end
    return result
end
```

We begin with the definition of `getGray` that will change the font color of the selected symbols from our game board. This should look nice on a standard, dark terminal display. Still, feel free to adjust the color to your needs (although if you use a terminal with a white background you may rather stop and get some help). Anyway, a field not taken by one of the players (`isFree2Take`) will be colored by `colorFieldNumbers`.

Personally, I would also opt to add the function for the triplets detection (`isTriplet`), which we will use to color them (first one we find based on `LINES`) with `colorFirstTriplet`. This should allow us for easier visual determination when the game is over (later on we will also use it in `isGameWon`).

[269]https://docs.julialang.org/en/v1/base/base/#Base.Enums.Enum

```
# https://en.wikipedia.org/wiki/ANSI_escape_code#Colors
function getRed(s::Str)::Str
    # "\x1b[31m" sets forground color to red
    # "\x1b[0m" resets forground color to default value
    return "\x1b[31m" * s * "\x1b[0m"
end

function isTriplet(v::Vec{Str})::Bool
    @assert length(v) == 3 "length(v) must be equal 3"
    return join(v) == "XXX" || join(v) == "OOO"
end

function colorFirstTriplet(board::Vec{Str})::Vec{Str}
    result::Vec{Str} = copy(board)
    for line in LINES
        if isTriplet(board[line])
            result[line] = getRed.(result[line])
            return result
        end
    end
    return result
end
```

Notice, that neither `colorFieldNumbers`, nor `colorFirstTriplet` modify the original game board, instead they produce a copy of it which is returned as a result (since the game board is a short vector there shouldn't be any serious performance issues).

Now, we are ready to print.

```
# https://en.wikipedia.org/wiki/ANSI_escape_code
function clearLines(nLines::Int)::Nothing
    @assert 0 < nLines "nLines must be a positive integer"
    # "\033[xxxA" - xxx moves cursor up xxx LINES
    print("\033[", nLines, "A")
    # "\033[0J" - clears from cursor position till the end of the screen
    print("\033[0J")
    return nothing
end

function printBoard(board::Vec{Str})::Nothing
    bd::Vec{Str} = colorFieldNumbers(board)
    bd = colorFirstTriplet(bd)
    for row in LINES[1:3] # first 3 LINES are for rows
        println(" ", join(bd[row], " | "))
        println("---+---+---")
    end
```

```
    clearLines(1)
    return nothing
end
```

First, we declare `clearLines`, it will help us to tidy the printout (e.g., while playing the game we will have to redraw the game board a couple of times). Next, we proceed with `printBoard`. We color the board with the previously defined functions and move row by row (`LINES[1:3]` contains the indices for the three rows). We `join` the contents of a row together (we glue them with `" | "`) and print it (`println`). We follow it by a row separator (`println("---+---+---")`). Once we're finished we remove the last row separator with `clearLines(1)` (we do not want it, but it was printed because we were too lazy to add an if statement in our for loop).

So far, so good, time to handle a human player's (aka user's) move.

```
function getUserInput(prompt::Str)::Str
    print(prompt)
    input::Str = readline()
    return strip(input)
end

function isMoveLegal(move::Str, board::Vec{Str})::Bool
    num::Int = 0
    try
        num = parse(Int, move)
    catch
        return false
    end
    return (num in eachindex(board)) && isFree2Take(board[num])
end

function getUserMove(gameBoard::Vec{Str})::Int
    input::Str = getUserInput("Enter your move: ")
    while !isMoveLegal(input, gameBoard)
        clearLines(1)
        input = getUserInput("Illegal move. Try again. Enter your move:
")
    end
    return parse(Int, input)
end
```

> Note. Using `while` loop always carries a risk of it being infinite, that's why it is worth to know that you can always press Ctrl+C[270] that should terminate the program execution.

We begin with `getUserInput` a function that takes the `prompt` (its argument, it tells the user what to do), prints it, and accepts the user's input (`readline`) that is returned as a result (after `striping` it from space/tab/new line characters that may be on the edges).

Next, we make sure that the move made by the user is legal (`isMoveLegal`), i.e. it can be correctly converted to an integer (`parse(Int, move)`), it is in the acceptable range (`num in eachindex(board)`) and is the field free to place the player's mark (`isFree2Take(board[num])`). Notice, the use of `try` and `catch` construct. First we `try` to make an integer out of the string obtained from the user (`parse(Int, move)`). This may fail (e.g., because we got the letter `"a"` instead of the number `"2"`). Such a failure, will result in an error that would normally terminate the program execution. We don't want that to happen, so we `catch` a possible error and instead of terminating the program, we just `return false`. If the `try` succeeds, we skip the `catch` part and go straight to the next statement after the `try-catch` block (`return (num in eachindex(board)) && isFree2Take(board[num])`) that we already discussed.

Finally, we declare `getUserMove`, a function that asks the user for a move and is quite persistent about it. If the user gives a correct move the first time (`input::Str = getUserInput("Enter your move: ")`) then the while loop condition (`!isMoveLegal(input, gameBoard)`) is false and the loop isn't executed at all (we move to the return statement). However, if the user plays tricks on us and wants to smuggle an illegal move (or maybe they just did it absent-mindedly) then the condition (`!isMoveLegal(input, gameBoard)`) is true and `while` it is we nag the user for a correct move (`"Illegal move. Try again. Enter your move: "`).

OK, and how about a computer move.

[270] https://en.wikipedia.org/wiki/Control-C

```
function getComputerMove(board::Vec{Str})::Int
    move::Int = 0
    for i in eachindex(board)
        if isFree2Take(board[i])
            move = i
            break
        end
    end
    println("Computer plays: ", move)
    return move
end
```

We start small, `getComputerMove` will simply walk through the board and return an index (`i`) of a first empty, i.e., not taken by a player (`isFree2Take(board[i])`) field. If all the fields are taken it will return `0` (in reality this will never happen as we will see in `playGame` later on). Since `getUserMove` prints one line of a screen output, then so does `getComputerMove` (`println("Computer plays: ", move)`) for compatibility.

Time to actually make a move that we obtained for a player.

```
function makeMove!(move::Int, player::Str, board::Vec{Str})::Nothing
    @assert move in eachindex(board) "move must be in range [1-9]"
    @assert player in PLAYERS "player must be X or O"
    if isFree2Take(board[move])
        board[move] = player
    end
    return nothing
end
```

For that we just take the `move`, a `player` for whom we place the mark, and the game `board` that we will modify. If a given field isn't taken (or to put it differently, it's free to take, hence `if isFree2Take(board[move])`) we just put the mark for a player there (`board[move] = player`).

Time to write `playMove`, a function that will handle a player, their move and its display on the screen.

```
function playMove!(player::Str, board::Vec{Str})::Nothing
    @assert player in PLAYERS "player must be X or O"
```

```
    printBoard(board)
    move::Int = (player=="X") ? getUserMove(board) :
getComputerMove(board)
    makeMove!(move, player, board)
    clearLines(6)
    printBoard(board)
    return nothing
end
```

We begin by displaying the board (`printBoard`) and obtaining a `move` for a player (`(player == "X") ? getUserMove(board) : getComputerMove(board)`). Once we got the move, we place the correct marker on the board (with `makeMove!`) and re-draw the board (`clearLines` and `printBoard`).

Now, we are almost ready to actually play a game. Almost, because we need a few more helper functions. First, we must figure out when the game is over and why. This can be simply achieved with the following snippet.

```
function isGameWon(board::Vec{Str})::Bool
    for line in LINES
        if isTriplet(board[line])
            return true
        end
    end
    return false
end

function isNoMoreMoves(board::Vec{Str})::Bool
    for i in eachindex(board)
        if isFree2Take(board[i])
            return false
        end
    end
    return true
end

function isGameOver(board::Vec{Str})::Bool
    return isGameWon(board) || isNoMoreMoves(board)
end
```

Once the game is over we display an appropriate info.

```
function displayGameOverScreen(player::Str, board::Vec{Str})::Nothing
    @assert player in PLAYERS "player must be X or O"
    printBoard(board)
    print("Game Over. ")
    isGameWon(board) ?
        println(player == "X" ? "You" : "Computer", " won.") :
        println("Draw.")
    return nothing
end
```

And finally, we're ready to play the game.

```
function togglePlayer(player::Str)::Str
    @assert player in PLAYERS "player must be X or O"
    return player == "X" ? "O" : "X"
end

function playGame()::Nothing
    board::Vec{Str} = getNewGameBoard()
    player::Str = "O"
    while !isGameOver(board)
        player = togglePlayer(player)
        playMove!(player, board)
        clearLines(5)
    end
    displayGameOverScreen(player, board)
    return nothing
end
```

Inside `playGame` we initialize `board` and the `player` on a move. Next, while the game isn't over (`while !isGameOver(board)`), we toggle the player (`togglePlayer(player)`), `playMove` and clear the display (`clearLines(5)`) before another move. When the game is finished we just `displayGameOverScreen`. And voila. You can `playGame`. Test it, e.g., with the following sequence of moves: 2, 3, 7, 6, 9 - you win; 2, 3, 6, 8 - computer wins; 7, 2, 4, 6, 9 - draw.

There are a couple of things to improve on (if you want to). For instance, you could add a `sleep` statement into `getComputerMove` so that the user got time to read the message with move declaration (`println("Computer plays: ", move)`). Moreover, as for now the algorithm generating move in `getComputerMove` is great for testing,

but gets boring pretty quickly, feel free to change it (or try to beat a slightly more challenging algorithm found in the code snippets[271]).

Lastly, like in Section 17.2 you could also add the functionality to run the game from a terminal (with `julia tic_tac_toe.jl`).

```
function main()::Nothing
    println("This is a toy program to play a tic-tac-toe game.")
    println("Note: your terminal must support ANSI escape codes.\n")

    # y(es) - default choice (also with Enter), anything else: no
    println("Continue with the game? [Y/n]")
    choice::Str = readline()
    if lowercase(strip(choice)) in ["y", "yes", ""]
        playGame()
    end

    println("\nThat's all. Goodbye!")

    return nothing
end

if abspath(PROGRAM_FILE) == @__FILE__
    main()
end
```

That's it. Have fun playing the game.

[271] https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/tic_tac_toe

Touch Typing

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[272](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[273].

Problem

In this exercise your job is to write a terminal[274]based application, similar to the one presented in the GIF below, that measures your (touch) typing speed.

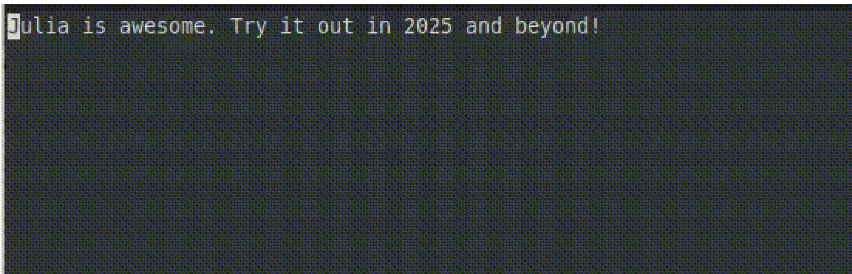


Figure 14: A terminal based application that measures typing speed (animation works only in an HTML document).

Your program should:

- mark the characters (in)correctly typed
- allow to delete the (incorrectly) typed characters
- display some basic summary of the speed (like WPM - words per minute)

[272]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/touch_typing

[273]https://docs.julialang.org/en/v1/stdlib/Pkg/

[274]https://en.wikipedia.org/wiki/Terminal_emulator

To make it easier, you may assume it to always operate on a short line of text (let's say 50 characters long) composed only of the characters from the standard Latin alphabet encoded by ASCII[275].

Solution

Let's approach the problem one step at a time. First, a formatting function `getColoredTxt`. The function will colorize the letters based on the correctness of our input. To that end we will reuse some of the code (see `getRed` and `getGreen` below) from the previous chapter (see Section 29.2).

```
# https://en.wikipedia.org/wiki/ANSI_escape_code#Colors
function getRed(c::Char)::Str
    return "\x1b[31m" * s * "\x1b[0m"
end

function getGreen(c::Char)::Str
    return "\x1b[32m" * s * "\x1b[0m"
end

function getColoredTxt(typedTxt::Str, referenceTxt::Str)::Str
    result::Str = ""
    for i in eachindex(referenceTxt)
        if i > length(typedTxt)
            result *= referenceTxt[i]
        elseif typedTxt[i] == referenceTxt[i]
            result *= getGreen(referenceTxt[i])
        else
            result *= getRed(referenceTxt[i])
        end
    end
    return result
end
```

> ***Note:*** In this chapter we rely on the assumption that we operate on a text composed of standard ASCII charset. Be aware that in the case of other charsets the indexing may not work as intended (see the docs[276]).

[275] https://en.wikipedia.org/wiki/ASCII

[276] https://docs.julialang.org/en/v1/manual/strings/#Unicode-and-UTF-8

The code is rather simple, we traverse the `referenceTxt`, i.e. the text we are suppose to type, with a `for` loop and indexing (`i`). If the text we already typed (`typedTxt`) is shorter than the current index (`i > length(typedTxt)`) we just append the character of the reference text to the `result` without coloring (`result *= referenceTxt[i]`). Otherwise we color the character of our `referenceTxt[i]` green (`getGreen`) in the case of a match (`typedTxt[i] == referenceTxt[i]`) or we color it red (`getRed`) otherwise. Finally, we `return` the colored text (`result`) for printout.

Now, in order to play our touch typing game we need a way to read a character or characters from the terminal[277]. This could be done with read[278]or with `readline` that we met in Section 29.2 . The problem is that by default, those are blocking functions (you need to press Enter for the `Char`/`String` to be read into your program). It turns out that an immediate, non-blocking readout in Julia isn't trivial to get. One option suggested by the Rosetta Code[279]website is to use an external library (the Gtk.jl presented in the link above seems to be no longer maintained). Other possibility would be to do this in a programming language better adjusted for such low level tasks, like C, and execute it from Julia (similarly to the suggestions found in this video[280]). However, in order to keep the solution minimal I will rely on `stty`, a terminal command found in UNIX(like) systems. If you don't have it on your computer you need to find some other way (or just skip this task).

> ***Note:*** Type `man stty` (and press Enter) into your terminal to check if you have the program installed on your system (`q` - closes the man page).

OK, let's play the game.

[277] https://en.wikipedia.org/wiki/Terminal_emulator
[278] https://docs.julialang.org/en/v1/base/io-network/#Base.read
[279] https://rosettacode.org/wiki/Keyboard_input/Flush_the_keyboard_buffer#Julia
[280] https://www.youtube.com/watch?v=obCMGkQ8Y8g

```
# more info on stty, type in the terminal: man stty
# display current stty settings with: stty -a (or: stty --all)
function playTypingGame(text2beTyped::Str)::Str
    c::Char = ' '
    typedTxt::Str = ""
    cursorCol::Int = 1
    run(`stty raw -echo`) # raw mode - reads single character
immediately
    while length(text2beTyped) > length(typedTxt)
        print("\r", getColoredTxt(typedTxt, text2beTyped))
        print("\x1b[", cursorCol, "G") # mv curs to cursorCol
        c = read(stdin, Char) # read a character without Enter
        typedTxt *= c
        cursorCol = length(typedTxt) + 1
    end
    println("\r", getColoredTxt(typedTxt, text2beTyped))
    run(`stty cooked echo`) # reset to default behavior
    return typedTxt
end
```

First, we declare and initialize a couple of variables that we will use later on: `c` to hold the character typed by the user, `typedTxt` to contain everything that the player typed and `cursorCol` which is a cursor position over a letter to be typed. Next, we execute a proper terminal command with run[281](notice the backticks). Now, for as long (`while`) as we haven't typed the whole `text2beTyped` (`length(text2beTyped) > length(typedTxt)`) we print the colored text (`\r` moves the cursor to the beginning of the line). Of course, we remember to set the cursor in the appropriate column (`"\x1b[", cursorCol, "G"`). Next we `read` a character typed by the player (`stdin` means standard input[282], it is a variable defined by Base[283]). Afterwords, we append the character (`c`) to the `typedTxt` and move the cursor by one column. Once we finish, we do some cleanup. We reprint the whole typed text and reset the terminal to its default values with `run`. We return `typedTxt` for further usage (by a summary function that will be defined soon).

The above is a reasonable approach, but there is a small problem with our `playTypingGame`. The raw mode that we use will turn off special

[281]https://docs.julialang.org/en/v1/base/base/#Base.run
[282]https://en.wikipedia.org/wiki/Standard_streams
[283]https://docs.julialang.org/en/v1/base/base/#Base

treatments of key-presses that we are accustomed to. For instance, currently there is no way to delete a character, nor terminate a program early with customary (Ctrl-C). I order to get this behavior we need to either turn off the raw mode or fix the problem ourselves.

```
function isDelete(c::Char)::Bool
    return c == '\x08' || c == '\x7F' # bacspace or delete
end

function isAbort(c::Char)::Bool
    return c == '\x03' || c == '\x04' # Ctrl-C or Ctrl-D
end

# more info on stty, type in the terminal: man stty
# display current stty settings with: stty -a (or: stty --all)
function playTypingGame(text2beTyped::Str)::Str
    c::Char = ' '
    typedTxt::Str = ""
    cursorCol::Int = 1
    run(`stty raw -echo`) # raw mode - reads single character
immediately
    while length(text2beTyped) > length(typedTxt)
        print("\r", getColoredTxt(typedTxt, text2beTyped))
        print("\x1b[", cursorCol, "G") # mv curs to cursorCol
        c = read(stdin, Char) # read a character without Enter
        if isDelete(c)
            typedTxt = typedTxt[1:(end-1)]
        elseif isAbort(c)
            break
        elseif isascii(c)
            typedTxt *= c
        else # do noting if user types e.g. ś ć or other strange chars
            nothing
        end
        cursorCol = length(typedTxt) + 1
    end
    println("\r", getColoredTxt(typedTxt, text2beTyped))
    run(`stty cooked echo`) # reset to default behavior
    return typedTxt
end
```

Much better. we just check for the hexadecimal (\x) ASCII code[284]for the specific characters. When a delete key is pressed we remove the last character from the typed text (`typedTxt[1:(end-1)]`). When an abort signal is send we just `break` the loop and leave early.

[284]https://pl.wikipedia.org/wiki/ASCII

Now, we add the summary statistics.

```
function getAccuracy(typedTxt::Str, text2beTyped::Str)::Flt
    len1::Int = length(typedTxt)
    len2::Int = length(text2beTyped)
    @assert len1 <= len2 "len1 must be <= len2"
    correctlyTyped::Vec{Bool} = Vec{Bool}(undef, len1)
    for i in eachindex(correctlyTyped)
        correctlyTyped[i] = typedTxt[i] == text2beTyped[i]
    end
    return sum(correctlyTyped) / length(correctlyTyped)
end

function printSummary(typedTxt::Str, text2beTyped::Str,
                      elapsedTimeSec::Flt)::Nothing
    wordLen::Int = 5 # avg. word length in English
    secsPerMin::Int = 60
    len1::Int = length(typedTxt)
    len2::Int = length(text2beTyped)
    cpm::Flt = len1 / elapsedTimeSec * secsPerMin
    wpm::Flt = cpm / wordLen
    acc::Flt = getAccuracy(typedTxt, text2beTyped)
    println("\n---Summary---")
    println("Elapsed time: ", round(elapsedTimeSec, digits=2), "
seconds")
    println("Typed characters: $len1/$len2")
    println("Characters per minute: ", round(cpm, digits=1))
    println("Words per minute: ", round(wpm, digits=1))
    println("Accuracy: ", round(acc * 100, digits=2), "%")
    return nothing
end
```

You can choose a different set of statistics, but I picked accuracy (% of characters that were typed correctly), number of characters per minute (`cpm`) and number of words per minute `wpm`.

As before (see Section 29.2) we finish with the main function.

```
function main()::Nothing

    println("Hello. This is a toy program for touch typing.")
    println("It should work well on terminals that: ")
    println("- support ANSI escape codes,")
    println("- got stty.\n")

    println("Press Enter (or any key and Enter) and start typing.")
    println("Press q and Enter to quit now.")
    choice::Str = readline()
```

```
    if lowercase(strip(choice)) != "q"
        txt2type::Str = "Julia is awesome. Try it out in 2025 and
beyond!"
        timeStart::Flt = time()
        typedTxt::Str = playTypingGame(txt2type)
        timeEnd::Flt = time()
        elapsedTimeSeconds::Flt = timeEnd - timeStart
        printSummary(typedTxt, txt2type, elapsedTimeSeconds)
    end

    println("\nThat's all. Goodbye!")

    return nothing
end

if abspath(PROGRAM_FILE) == @__FILE__
    main()
end
```

And voila, the task was completed in like a hundred lines of code or so. You may now open your terminal, type: `julia touch_typing.jl` and test your typing speed.

If you still haven't had enough then feel free to extend the program so that it can also handle a bit longer, multi-line texts. Alternatively, you may examine such a program in the code snippets[285].

[285] https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/touch_typing

Compound Interest

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[286](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[287].

Problem

There is this cartoon Futurama[288]that tells a story of a guy named Fry. The character was accidentally frozen on December 31, 1999 and wakes up back to life on December 31, 2999. In season 1 episode 6 Fry visits his old bank. The teller informs him that he had on his account 93 cents in 1999. However, with an average yearly interest of 2.25% over the period of a thousand years it gives $4.3 billion.

Read about compound interest[289]and use Julia to answer a few questions.

Question 1

Is Fry really a billionaire?

Question 2

According to this page[290](careful it's in Polish) the inflation in Poland over the period 2020-2024 was: 3.4%, 5.1%, 14.4%, 11.4%, and 3.6%. If on December 31, 2019 my monthly salary was $10,000 (I wished) then

[286]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/compound_interest

[287]https://docs.julialang.org/en/v1/stdlib/Pkg/

[288]https://en.wikipedia.org/wiki/Futurama

[289]https://en.wikipedia.org/wiki/Compound_interest

[290]https://pl.wikipedia.org/wiki/Inflacja_w_Polsce#Historia

how much I would have to earn in January 2025 to be able to buy the same amount of goods like in 2019?

Question 3

Imagine that on January 1, 2020 I opened a bank deposit for 5 years with a yearly interest rate of 6%. Given the inflation rates from the question 2 (Section 31.1.2) would I make any real profit on January 1, 2025?

Question 4

Let's say that an average Polish male, aka Jan Kowalski, retires at the age of 65, while his life expectancy at birth is 75 years. Jan starts his first job at the age of 20 and earns $3,000 a month. However, since he had heard that the pension a person receives is equal to 50% of their last salary, he decided to save $200 monthly (a bit less than 7% of his earnings). Once he retires he'll take the missing 50% from the pile of money that he saved so that quality of his life will not change.

Assume that: 1) there is no inflation; 2) Jan's salary is constant throughout his lifetime; 3) he pays $2,400 into his savings account at the beginning of each year; 4) the account gives him 2% yearly. Tell roughly how long will the money last on the retirement?

Solution

Before we begin a warning. The following section, calculations, etc. may or may not be accurate and are only meant as a programming exercise, not a financial advice.

As a prelude let's write a helper function that will nicely display the amount of money we get and improve the readability of our results.

```
function getFormattedMoney(money::Real, sep::Char=','))::Str
    @assert money >= 0 "money must be >= 0"
    amount::Str = round(Int, money) |> string
    result::Str = ""
    counter::Int = 0
    for digit in reverse(amount) # digit is a single digit (type Char)
        if counter == 3
            result = sep * result
            counter = 0
```

```
        end
        result = digit * result
        counter += 1
    end
    return result * " USD"
end
```

The function may not be the most performant, but it was pretty easy to write. It receives money as a Real[291]number and sets apart every three digits with a separator (`sep`) of our choice. Since in English a decimal separator is `.` (dot) and a thousand separator is `,` (comma) then that's what we used here as our default (`sep::Char=','`). Inside our formatter we round the number to integer (`round(Int, money)`) and convert it to `string`. Next we traverse all the digits (`for digit`) in the opposite direction (from right to left) thanks to the `reverse(amount)`. Every third digit (`if counter == 3`) we place our `sep` to the `result` and reset the counter (`counter = 0`). Besides, we prepend our `digit` to the `result` (`digit * result`) and increase the counter (`counter +=1`). In the end we return the formatted number (`result * " USD"`).

> ***Note:*** For another way to format a number to currency see Section 27.2.4 .

Let's see how it works.

```
getFormattedMoney(12345.06)
```

12,345 USD

I think that twelve thousand three hundred and forty five is much easier to process this way than in its alternative transcription form (`12345`). Still, be aware of the rounding that takes place in it.

[291]https://docs.julialang.org/en/v1/base/numbers/

Answer 1

Before we begin a quick refresher on percentages. As explained, e.g. here[292]per definition a percentage is a hundredth part of the whole and can be represented in a few equivalent forms, i.e.

- 0% = $\frac{0}{100}$ = 0/100 = 0.00 = 0,
- 5% = $\frac{5}{100}$ = 5/100 = 0.05,
- 20% = $\frac{20}{100}$ = 20/100 = 0.20 = 0.2,
- 75% = $\frac{75}{100}$ = 75/100 = 0.75,
- 100% = $\frac{100}{100}$ = 100/100 = 1.00 = 1,
- 105% = $\frac{105}{100}$ = 105/100 = 1.05,
- 110% = $\frac{110}{100}$ = 110/100 = 1.10 = 1.1, etc.

For calculations it is especially useful to use them as decimals, hence we will often divide a percentage by one hundred. Now, let's say that I got $100 (this initial sum of money is called the **principal**) on a deposit that pays 5% interest yearly. That means that after one year I will get $105 or 105% of my initial principal (the 5% of $100 is $5, and it is the interest we get for giving our money to the bank). I can express this mathematically as `$100 * 105% = $105` or to make it easier to type with a calculator `100 * 1.05 = 105`. If the deposit lasted two years then I would have to repeat the process one more time for year number two, i.e. `$105 * 105% = $110.25` (105% of my new principal), also to be expressed as `105 * 1.05 = 110.25`. Since as we said the `$105` is actually `100 * 1.05` then I can rewrite it as `(100 * 1.05) * 1.05 = 110.25`. For year number three I got `$110.25 * 105% = $115.7625` or `110.25 * 1.05 = 115.7625` or `((100 * 1.05) * 1.05) * 1.05 = 115.7625`. The parenthesis are there to signify boundaries of mathematical operations for a previous year. We can get rid of them to get: `100 * 1.05 * 1.05 * 1.05` (we multiply `1.05` by itself as many times as there are years). Hence a pattern emerges:

$$deposit\ value = \$100 * 1.05^{number\ of\ years}$$

[292] https://b-lukaszuk.github.io/RJ_BS_eng/statistics_intro_probability_definition.html

or more generally (remember about the order of mathematical operations):

$$total\ value = initial\ principal * (1 + percentage/100)^{number\ of\ years}$$

Let's put that into a Julia's function.

```
function getValue(principal::Real, avgPercentage::Real, years::Int)::Flt
    @assert years > 0 "years must be greater than 0"
    @assert principal > 0 "principal must be greater than 0"
    return principal * (1+avgPercentage/100)^years
end
```

And test it on our previous example.

```
round.([getValue(100, 5, i) for i in 1:3], digits=4)
```

```
[105.0, 110.25, 115.7625]
```

Now we are ready to see if Fry is a billionaire.

```
# Futurama s1e6: 93 cents ($0.93), 2.25%, 1_000 years
frysMoney = getValue(0.93, 2.25, 1_000)
frysMoney |> getFormattedMoney
```

4,283,508,450 USD

And indeed he is (four billion two hundred eighty three million, etc.). Still, before you head towards cryogenic clinic to duplicate Fry's success be aware of all the issues with cryonics[293]and that most of the people frozen in 1960s' are not preserved today. Therefore, the profit seems to be purely theoretical.

Anyway, as said the function could be used to estimate the nominal value that you would get from your deposit (with a stable yearly interest rate). You could also look at it from the other end and estimate how much you would have to earn to cover up for an average inflation rate over a few years. Keep that in mind as we go to the solution for Question 2 (see Section 31.1.2).

[293]https://en.wikipedia.org/wiki/Cryonics#History

Answer 2

What about Question 2 (Section 31.1.2). If on December 31, 2019 my monthly salary was $10,000 then how much I would have to earn in January 2025 to maintain my purchasing power given that the inflation rates for years 2020-2024 were equal: 3.4%, 5.1%, 14.4%, 11.4%, and 3.6%. To answer that we will use the same reasoning as in the previous section (Section 31.2.1), but since the percentages differ from year to year we will use a `for` loop to cover for that.

```
function getValue(principal::Real, percentages::Vec{<:Real})::Flt
    @assert principal > 0 "principal must be greater than 0"
    for p in percentages
        principal *= 1 + (p / 100)
    end
    return principal
end
```

Second `getValue` method defined, time to put it into good use.

```
money2019 = 10_000 # Dec 31, 2019
inflPoland = [3.4, 5.1, 14.4, 11.4, 3.6] # yrs: 2020-2024
money2025infl = getValue(money2019, inflPoland) # Jan 1, 2025
(
    getFormattedMoney(money2019),
    getFormattedMoney(money2025infl)
)
```

```
("10,000 USD", "14,348 USD")
```

So I would need to earn roughly $14,348 to be able to buy the same amount of goods in 2025 as I could with $10,000 in 2019. Not sure why, but that doesn't put a smile on my face.

Answer 3

As a final step let's think was it worth a while to open a 5 years long bank deposit (6% yearly interest rate) on January 1, 2020 given the inflation rates discussed in the previous section (Section 31.2.2). Would I make any real profit on January 1, 2025?

In order to figure that out we need to counterbalance two factors. The increase in the nominal value that we get due to the interest rate and a drop in the real value of money due to the inflation rate. In other words, we might want to calculate the real percentage change in money value given the two factors combined. For that we should either implement a suitable formula obtained from a reliable source or come up with the one ourselves. In order to learn we will try the latter approach (no pain, no gain). Still, we may make a mistake so, take it with a grain of salt.

Anyway, I think we will use proportions (I believe they taught me that in the high school in chemistry, so bear with me, you can do it) and some simple imaginable example. To make sure we are on the same page let's talk briefly about the proportions.

Imagine that for $10 (`usd` below) we can buy 2 bread loafs (`bl` below). If so, then how much bread can we buy for $5? That's easy, one bread loaf. You can solve it with proportions like so:

$$10\,usd - 2\,bl \qquad (10)$$

$$5\,usd - x\,bl \qquad (11)$$

We set the same units on the same sides (left - right). Next, in order to get x we multiply the numbers on the diagonals: 5 * 2 goes to the numerator and 10 goes into denominator (since it got no pair on the diagonal it falls to the bottom):

$x = \frac{5\,usd * 2\,bl}{10\,usd}$

Then we forward to our solution.

$x = \frac{5*2}{10} = \frac{10}{10} = 1$

This works because Equation 10 and Equation 11 could be rewritten as:

$\frac{10}{2} = \frac{5}{x}$

or

$\frac{2}{10} = \frac{x}{5}$

The above is basically just finding an equivalent fraction that we learned in our primary schools (how many 5ths is $\frac{2}{10}$?). Still, I like and remember the proportions example better.

With that under our belts let's follow with a simple example. Imagine that in this year for $100 (`usd` below) we can buy 100 chocolate bars (`cb` below). Due to the yearly inflation that is 2%, the same 100 chocolate bars will cost $102 after a year. Luckily, thanks to the 5% yearly interest rate we will have $105 in banknotes from our deposit. So how many chocolate bars will we be able to buy after a year?

That's easy thanks to the proportions.

$$102\,usd - 100\,cb \qquad (12)$$

$$105\,usd - x\,cb \qquad (13)$$

Which gives us:

$x = \frac{105\,usd * 100\,cb}{102\,usd}$

and

$x = \frac{105*100}{102} = calculator\ does\ pip, pip, \ldots \approx 102.94$

Therefore, we can see that in this scenario the $105 in banknotes will allow us to buy 102.94 chocolate bars (we think of it as a stable product that by itself does not gain or loose value over time). Based on the chocolate bars we can evaluate the real gain/loss of our nominal money to be: `102.94 - 100 = 2.94 chocolate bar` or 2.94% (chocolate bars were just an abstraction needed as a reference point, something that does not change its value over time, hence a constant). When we put it all together (starting from Equation 12) in a formula, we get (notice that the 100 below is a constant that we referred to in the sentence before):

$$real\ percentage = \frac{105\,usd * 100}{102\,usd} - 100 \qquad (14)$$

Just like the chocolate bars also the dollars are a placeholder in our example. We used them because the more material and concrete the

objects are the easier it is to think about them and manipulate them in our heads. Notice that `105 usd` in Equation 14 and Equation 13 actually stands for percentage gain in value (`100 + inflation rate`) of our money (the 105% that we used in our explanation in the calculations in Section 31.2.1). On the other hand, `102 usd` in Equation 14 and Equation 12 is actually a placeholder for percentage change in value due to the inflation (we divide by it, so we decrease our gain in numerator by it). Therefore we can rewrite Equation 14 to:

$$real\,percentage = \frac{(100 + interest\,rate)*100}{100 + inflation\,rate} - 100 \qquad (15)$$

Now let's put Equation 15 into a Julia's function.

```
function getRealPercChange(interestPerc::Real, inflPerc::Real)::Flt
    return ((100 + interestPerc) * 100) / (100 + inflPerc) - 100
end
```

> ***Note:*** As a practical exercise you may further simplify the formula in Equation 15 using a pen and paper. Compare your result with the output of Symbolics.jl (that we met in Section 3.2), e.g. `Sym.@variables realPerc interPerc inflPerc` and `Sym.simplify(realPerc ~ ((100 + interPerc)*100)/(100 + inflPerc)-100)`. Of course, if you think that's too much mathematics for one day, then don`t. Julia won't mind computing the result of Equation 15 for you.

Now we can use it to write our final, third method, for `getValue`.

```
function getValue(principal::Real,
                  interestPercs::Vec{<:Real},
                  inflationPercs::Vec{<:Real})::Flt
    @assert principal > 0 "principal must be greater than 0"
    @assert(length(interestPercs) == length(inflationPercs),
            "interestPercs and inflationPercs must be of equal lenghts")
    for (intr, infl) in zip(interestPercs, inflationPercs)
        principal = getValue(principal, getRealPercChange(intr, infl),
1)
    end
```

```
    return principal
end
```

We will use it to answer our question.

```
interestDeposit = 6 # yrs: 2020-2025
numYrs = length(inflPoland)
money2025deposit = getValue(money2019,
    interestDeposit, numYrs)
money2025depositInflation = getValue(
    money2019,
    repeat([interestDeposit], numYrs), inflPoland)


(
    getFormattedMoney(money2019), # Dec 31, 2019 or Jan 1, 2020
    getFormattedMoney(money2025deposit), # Jan 1, 2025
    getFormattedMoney(money2025depositInflation) # Jan 1, 2025
)
```

```
("10,000 USD", "13,382 USD", "9,327 USD")
```

And again, reality turns out to be disappointing. The initial principal of $10,000 (January 1, 2020) was increased by 6% yearly which gave me $13,382 in banknotes on January 1, 2025 for which I can buy the same amount of goods that I could for $9,327 on January 1, 2020. So despite more money in my wallet (nominal increase) I actually lost some real value. Eh.

OK, let's try to be optimists here. The glass is half full. By putting the money on the deposit (6% yearly) we lost some value. Still, if we left it on an ordinary account with a lousy 0.5% interest rate the financial outcome would be even more unsatisfactory.

```
(
    # nominal gain
    getFormattedMoney(
        getValue(money2019, 0.5, numYrs)),
    # real value
    getFormattedMoney(
        getValue(money2019, repeat([0.5], numYrs), inflPoland))
)
```

```
("10,253 USD", "7,146 USD")
```

So always look on the bright side of life, but look for something better and think before you leap.

Answer 4
Time for the last question.

Jan Kowalski saves $2,400 per year with constant 2% interest rate over his working life. If so, then how much will he get in the end and for how long will it last?

First, the savings.

```
function getRetirementSavings(moneyPerYr::Real, avgPercPerYr::Real,
                              years::Int)::Flt
    @assert moneyPerYr > 0 "moneyPerYr must be greater than 0"
    @assert avgPercPerYr > 0 "avgPercPerYr must be greater than 0"
    @assert 0 < years < 50 "years must be in range [1-49]"
    savings::Flt = 0.0
    multiplier::Flt = 1 + avgPercPerYr / 100
    for _ in 1:years
        savings += moneyPerYr # saving on Jan 1
        savings *= multiplier # interest paid on Dec 31
    end
    return savings
end
```

After the previous sections, the function may seem ridiculously simple to you. For each year (`for _ in 1:years`) we save some money (`savings += moneyPerYr`) of which we get our interest (`savings *= mulitplier`) based on the formulas discussed in (Section 31.2.1 and Section 31.2.2). Once we are done, we return what we saved.

```
savingsPerYr = 2_400 # or 200 * 12
savingsPercentageYr = 2
yrsWorking = 65 - 20

moneyPutAside = savingsPerYr * yrsWorking
savings = getRetirementSavings(
    savingsPerYr, savingsPercentageYr, yrsWorking)

(
```

```
    moneyPutAside |> getFormattedMoney,
    savings |> getFormattedMoney
)
```

```
("108,000 USD", "175,993 USD")
```

So over the period of 45 working years (`yrsWorking`) Jan Kowalski would have put aside roughly $108,000 which together with interests should give him $175,993.

Now, remember, that the idea is to take $1,500 of this pile of money every month to fill the gap caused by his pension being equal to 50% of his final salary (which we said was $3,000). So let's see for how long will it last.

```
# 1,500 usd a month, 12 months per year
round(savings / 1_500 / 12, digits=2)
```

9.78

He should be able to maintain his standard of living for roughly 10 years, which is his expected life expectancy on retirement (retires at 65, lives up to 75). So, I guess that if Jan intends to beat the averages and spend more time on the retirement, then he must save some more cash. The same seems to be true if the pension is less than 50% of his last salary. Anyway, it seems that to have some spare money on your retirement might be a prudent idea.

Mortgage

In this chapter I used the following libraries.

```
import CairoMakie as Cmk # external library
```

Still, once you read the problem description you may decide to do otherwise.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[294](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[295].

Problem

Sooner or later most of us ordinary folk end up taking a mortgage to buy an apartment or a house.

So here are two scenarios for you:

1. you borrow $200,000 (principal) for 20 years at 6.49% (constant yearly interest rate)
2. you borrow $200,000 (principal) for 30 years at 4.99% (constant yearly interest rate)

Write a Julia program that will tell you:

1. How much money will you pay every month (installment) in both cases (assume fixed rate mortgage)?
2. How much principal you will still owe to the bank at year 15 in each case?
3. When will your debt be ≤ $100'000?
4. Which of the two mortgages is more worth it for you (the smaller total cost and total interest you pay to the bank)?

[294]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/mortgage
[295]https://docs.julialang.org/en/v1/stdlib/Pkg/

Feel free to add some visual flair to your solution.

BTW. You may read about mortgages, e.g. here[296].

Solution

Before we begin a few short definitions that we will use here (so that we are on the same page).

- principal - the money you borrow form a bank
- interest - the money you pay extra (except for the money you borrowed)
- interest rate - a percentage of principal which you will pay as interest
- installment - fixed monthly payment to the bank in order to cover you debt, part of it goes to pay off the principal and part to pay off the interest

Next, let's use the formatting function from Section 31.2 .

```
# getFormattedMoney from chapter: compound interest, modified
function fmt(money::Real, sep::Char=',',)::Str
    @assert money >= 0 "money must be >= 0"
    amount::Str = round(Int, money) |> string
    result::Str = ""
    counter::Int = 0
    for digit in reverse(amount) # digit is a single digit (type Char)
        if counter == 3
            result = sep * result
            counter = 0
        end
        result = digit * result
        counter += 1
    end
    return result * " USD"
end
```

> ***Note:*** For another way to format a number to currency see Section 27.2.4 .

[296]https://en.wikipedia.org/wiki/Mortgage_calculator#

Time to define a `struct` that will contain the data necessary to perform calculations for a given mortgage.

```
struct Mortgage
    principal::Real
    interestPercYr::Real
    numMonths::Int

    Mortgage(p::Real, i::Real, n::Int) = (
        p < 1 || i < 0 || n < 12 || n > 480) ?
        error("incorrect field values") : new(p, i, n)
end

mortgage1 = Mortgage(200_000, 6.49, 20*12)
mortgage2 = Mortgage(200_000, 4.99, 30*12)
```

Finally, we are ready to calculate our monthly payment to the bank.

```
# calculate c - money paid to the bank every month
function getInstallment(m::Mortgage)::Flt
    p::Real = m.principal
    r::Real = m.interestPercYr / 100 / 12
    n::Int = m.numMonths
    if r == 0
        return p / n
    else
        numerator::Flt = r * p * (1+r)^n
        denominator::Flt = ((1 + r)^n) - 1
        return numerator / denominator
    end
end
```

All the formulas are based on this Wikipedia page[297]. Notice that the function's arguments (`Mortgage` fields in `getInstallment` and the arguments to `getPrincipalAfterMonth` below) contain longer (more descriptive) names, whereas inside the functions we use the abbreviations (case insensitive) found in the above-mentioned formulas .

With that done, we can answer how much money we will have to pay to the bank every month for the duration of our mortgage.

[297] https://en.wikipedia.org/wiki/Mortgage_calculator#

```
(
    getInstallment(mortgage1) |> fmt,
    getInstallment(mortgage2) |> fmt
)
```

```
("1,490 USD", "1,072 USD")
```

The money we still owe to the bank (principal) will change month after month (because every month we pay off a fraction of it with our installment), so let's calculate that.

```
# amount of money owed after every month
function getPrincipalAfterMonth(prevPrincipal::Real,
                                interestPercYr::Real,
                                installment::Flt)::Flt
    @assert((prevPrincipal >= 0 && interestPercYr >= 0 && installment >
0),
            "incorrect argument values")
    p::Real = prevPrincipal
    r::Real = interestPercYr / 100 / 12
    c::Flt = installment
    return (1 + r) * p - c
end
```

Not much to explain here, just a simple rewrite of the formula into Julia's code (first we increase the principal by interest, then we subtract the installment from it). Now we can estimate the principal owed to the bank each year.

```
# paying off mortgage year by year
# returns principal still owed every year
function getPrincipalOwedEachYr(m::Mortgage)::Vec{Flt}
    monthlyPayment::Flt = getInstallment(m)
    curPrincipal::Real = m.principal
    principalStillOwedYrs::Vec{Flt} = [curPrincipal]
    for month in 1:m.numMonths
        curPrincipal = getPrincipalAfterMonth(
            curPrincipal, m.interestPercYr, monthlyPayment)
        if month % 12 == 0
            push!(principalStillOwedYrs, curPrincipal)
        end
    end
    return principalStillOwedYrs
end
```

Here we calculate the principal owed after every month, and once a year (`month % 12 == 0`) we push it to a vector tracking its yearly change (`principalStillOwedYrs`). Let's see how it works, if we did it right then in the end the principal should drop to zero (small rounding errors possible).

```
principals1 = getPrincipalOwedEachYr(mortgage1)
principals2 = getPrincipalOwedEachYr(mortgage2)

(
    principals1[end],
    principals2[end],
    round(principals1[end], digits=2),
    round(principals2[end], digits=2)
)
```

```
(-1.1685642675729468e-8, -1.4030092643224634e-8, -0.0, -0.0)
```

So how much principal we still owe to the bank at the beginning of year 15 in each scenario?

```
(
    principals1[15] |> fmt,
    principals2[15] |> fmt
)
```

```
("88,661 USD", "141,638 USD")
```

Hmm, quite a lot (remember we borrowed $200,000 for 20 and 30 years). And when will this value drop $\leq$ 100,000 USD ($\leq$ half of what we borrowed)?

```
(
    findfirst(p -> p <= 100_000, principals1),
    findfirst(p -> p <= 100_000, principals2)
)
```

```
(15, 22)
```

In general, for quite some time the money we pay to the bank mostly pay off the interest and not the principal, so that it drops slowly at first (see Figure 15).

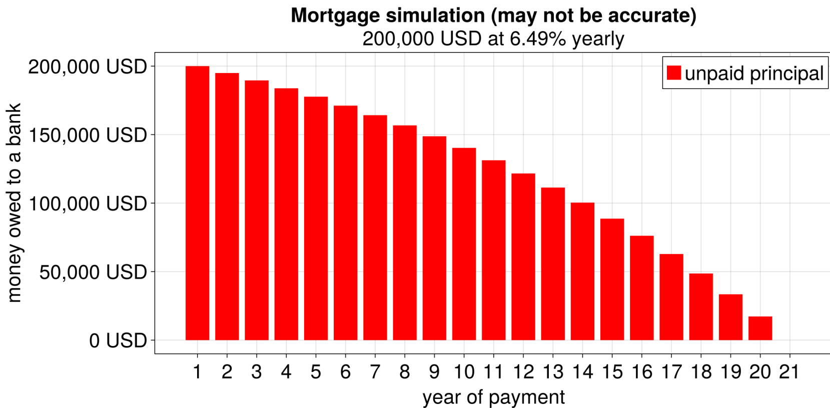


Figure 15: Principal still owed to the bank year by year. Mortgage: $200,000 at 6.49% yearly for 20 years. The estimation may not be accurate.

To answer the last question (which mortgage is more worth it for us in terms of total payment and total interest) we'll use a pie chart[298].

```
import CairoMakie as Cmk

function addPieChart!(m::Mortgage, fig::Cmk.Figure,
                      ax::Cmk.Axis, col::Int)::Nothing
    installment::Flt = getInstallment(m)
    totalInterest::Flt = installment * m.numMonths - m.principal
    yrs::Flt = round.(m.numMonths / 12, digits=2)
    colors::Vec{Str} = ["coral1", "turquoise2", "white", "white",
"white"]
    lebels::Vec{Str} = ["interest = $(fmt(totalInterest))",
                        "principal = $(fmt(m.principal))",
                        "$(yrs) years, $(m.interestPercYr)% yearly",
                        "total cost = $(fmt(installment *
m.numMonths))",
                        "monthly payment = $(fmt(installment))"]
    Cmk.pie!(ax, [totalInterest, m.principal], color=colors[1:2],
             radius=4, strokecolor=:white, strokewidth=5)
    Cmk.hidedecorations!(ax)
```

[298] https://docs.makie.org/v0.21/reference/plots/pie

```
    Cmk.hidespines!(ax)
    Cmk.Legend(fig[3, col],
               [Cmk.PolyElement(color=c) for c in colors],
               lebels, valign=:bottom, halign=:center, fontsize=60,
               framevisible=false)
    return nothing
end
```

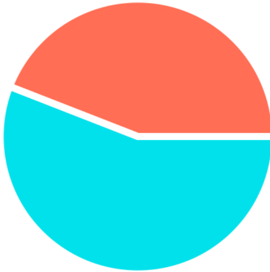
The function is rather simple. It adds a pie chart to an existing figure and axis. A point of notice, the colors used are `["coral1", "turquoise2", "white", "white", "white"]`. The first two will be used to paint the circle. But all of them, will be used in the legend (`Cmk.Legend`). Hence, we used `"white"` for the values that are not in the circle (white color on a white background in the legend is basically invisible). We also used string interpolation[299]where a simple interpolated value is placed after the dollar character (`$`) or in a more complicated case (a structure field, a calculation) it is put after the dollar character and within parenthesis (e.g. `"$(2*3)"`).

Time to draw a comparison

```
function drawComparison(m1::Mortgage, m2::Mortgage)::Cmk.Figure
    fig::Cmk.Figure = Cmk.Figure(fontsize=18)
    ax1::Cmk.Axis = Cmk.Axis(
        fig[1:2, 1], title="Mortgage simulation\n(may not be accurate)",
        limits=(-5, 5, -5, 5), aspect=1)
    ax2::Cmk.Axis = Cmk.Axis(
        fig[1:2, 2], title="Mortgage simulation\n(may not be accurate)",
        limits=(-5, 5, -5, 5), aspect=1)
    Cmk.linkxaxes!(ax1, ax2)
    Cmk.linkyaxes!(ax1, ax2)
    addPieChart!(m1, fig, ax1, 1)
    addPieChart!(m2, fig, ax2, 2)
    return fig
end

drawComparison(mortgage1, mortgage2)
```

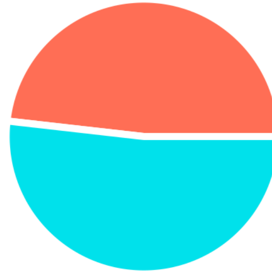
[299]https://docs.julialang.org/en/v1/manual/strings/#string-interpolation

Mortgage simulation
(may not be accurate)



interest = 157,593 USD
principal = 200,000 USD
20.0 years, 6.49% yearly
total cost = 357,593 USD
monthly payment = 1,490 USD

Mortgage simulation
(may not be accurate)



interest = 186,072 USD
principal = 200,000 USD
30.0 years, 4.99% yearly
total cost = 386,072 USD
monthly payment = 1,072 USD

Figure 16: Comparison of two mortgages (may not be accurate).

So it turns out that despite the higher interest rate of 6.49% overall we will pay less money to the bank for `mortgage1`. Therefore, if we are OK with the greater monthly payment (installment) then we may choose that one.

Of course, all the above was just a programming exercise, not a financial advice. Moreover, the simulation is likely to be inaccurate (to a various extent) for many reasons. For instance, a bank may calculate the interest every day, and not every month, in that case you will pay more. Compare with the simple example below and the compound interest from Section 31.2.1 .

```
# 6% yearly, after 1 year
(
    200_000 * (1 + 0.06) |> fmt, # capitalized yearly
    200_000 * (1 + (0.06/12))^12 |> fmt, # capitalized monthly
    200_000 * (1 + (0.06/365))^365 |> fmt, # capitalized daily
)
```

```
("212,000 USD", "212,336 USD", "212,366 USD")
```

Anyway, once you know what's going on, it should be easier to modify the program to reflect a particular scenario more closely.

Overpayment

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[300](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[301].

Problem

We finished the previous chapter (Section 32) with the discussion on a topic of a mortgage. In general, banks allow their clients to overpay their mortgages, which should be beneficial to the borrowers. So here is a task for you.

Write a computer program, that will estimate the savings you make by overpaying a mortgage (assume the whole overpayment goes to pay off the principal and reduces the number of installments).

Use the program to answer a few questions.

How much money can you expect to save in the case of `mortgage1` ($200,000, 6.49%, 20 years) and `mortgage2` ($200,000, 4.99%, 30 years) (see Section 32.2) if you overpay them regularly every month with $200.

For `mortgage1` which one is more worth it: to overpay it every month with $200 or to overpay it only once, let's say in month 13, with $20,000?

Which scenario would you expect to give you more profit (in nominal value): to overpay `mortgage1` with $20,000 in month 13, or to put this

[300]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/overpayment

[301]https://docs.julialang.org/en/v1/stdlib/Pkg/

$20,000 into a bank deposit that pays 5% yearly for the 19 years (roughly the remaining duration of the mortgage)?

Solution

There's no need to (completely) reinvent the wheel so we will use `Mortgage`, `fmt` and `getInstallment` we developed earlier (see Section 32.2). Just in case be sure to also check the terminology we defined there. Anyway, the first function we'll define in this chapter is `payOffMortgage`

```
# single month payment of mortgage, returns
# (remainingPrincipal, pincipalPaid, interestPaid)
function payOffMortgage(
    m::Mortgage, curPrincipal::Real, installment::Real,
    overpayment::Real)::Tuple{Real, Real, Real}
    interestDecimalMonth::Real = m.interestPercYr / 100 / 12
    interestPaid::Real = curPrincipal * interestDecimalMonth
    principalPaid::Real = installment - interestPaid
    newPrincipal::Real = curPrincipal - principalPaid
    return (newPrincipal - overpayment,
            principalPaid + overpayment, interestPaid)
end
```

The function accepts (among others) `curPrincipal`, `installment` and `overpayment` and does a payment for a single month. To that end, first we calculate the monthly interest rate as a decimal (`interestDecimalMonth`) and use it to calculate the interest and principal paid this month (`interestPaid` and `principalPaid`). Afterwards we calculate our the new, lower principal (`newPrincipal`). Finally, we return a tuple with 3 values: 1) the remaining principal (after a month), 2) principal paid in a given month (from `installment` and `overpayment`), and 3) interest paid (from `installment`). The remaining principal is `newPrincipal - overpayment`. The principal that we paid off this month is `principalPaid` and `overpayment`. The interest paid this month is just `interestPaid`.

Right away we see a reason or two why our function is likely not to be accurate. For once, we lack the rounding of money to 2 decimal points (as a bank would do). Secondly, a bank may charge a fee (or some money named otherwise) for every overpayment we make. Still, since

all this section is just a programming exercise and not a financial advice then we will not be bothered by that fact.

Still, we will improve our `payOffMortgage` a bit, by dealing with some edge cases: 1) when `curPrincipal` is 0 or negative (`if curPrincipal <= 0.0` below), 2) when `curPrincipal` is equal to or smaller than the principal paid in installment (`if curPrincipal <= principalPaid` below), and 3) when (`if newPrincipal <= overpayment` below). Therefore, our `payOffMortgage` will look something like:

```
# single month payment of mortgage
# (remainingPrincipal, pincipalPaid, interestPaid)
function payOffMortgage(
    m::Mortgage, curPrincipal::Real, installment::Real,
    overpayment::Real)::Tuple{Real, Real, Real}
    if curPrincipal <= 0.0
        return (curPrincipal, 0.0, 0.0)
    end
    interestDecimalMonth::Real = m.interestPercYr / 100 / 12
    interestPaid::Real = curPrincipal * interestDecimalMonth
    principalPaid::Real = installment - interestPaid
    if curPrincipal <= principalPaid
        return (0.0, curPrincipal, interestPaid)
    end
    newPrincipal::Real = curPrincipal - principalPaid
    if newPrincipal <= overpayment
        return (0.0, newPrincipal + principalPaid, interestPaid)
    end
    return (newPrincipal - overpayment,
            principalPaid + overpayment, interestPaid)
end
```

Once we can pay it off for a month, we can pay it off completely and summarize it. First, summary:

```
struct Summary
    principal
    interest
    months

    Summary(p::Real, i::Real, m::Int) = (
        p < 1 || i < 0 || m < 12 || m > 480) ?
        error("incorrect field values") : new(p, i, m)
end
```

Now, for the complete mortgage pay off.

```
# pay off mortgage fully, with overpayment
function payOffMortgage(
    m::Mortgage,
    overpayments::Dict{Int, <:Real})::Summary
    installment::Real = getInstallment(m) # monthly payment
    princLeft::Real = m.principal
    princPaid::Real = 0.0
    interPaid::Real = 0.0
    totalPrincPaid::Real = 0.0
    totalInterestPaid::Real = 0.0
    months::Int = 0
    for month in 1:m.numMonths
        if princLeft <= 0
            break
        end
        months += 1
        princLeft, princPaid, interPaid = payOffMortgage(
            m, princLeft, installment, get(overpayments, month, 0))
        totalPrincPaid += princPaid
        totalInterestPaid += interPaid
    end
    return Summary(totalPrincPaid, totalInterestPaid, months)
end

# pay off mortgage according to the schedule, no overpayment
function payOffMortgage(m::Mortgage)::Summary
    return payOffMortgage(m, Dict{Int, Real}())
end
```

We begin, by defining a few variables. For instance, `princLeft` will hold principal sill left to pay after a month, `princPaid` will contain the value of principal paid off in a given month, `intrPaid` will store the interest paid to a bank in a given month. The above will be used in the `for` loop and will change month by month. Next, the variables that will be used in the `Summary` struct returned by our function (`totalInterestPaid`, `totalInterestPaid`, `months`). In the `for` loop we pay off the mortgage month by month. If there is no principal to be paid off (`if princLeft <= 0`) we `break` early. Otherwise, we update the number of `months`, `totalPrincPaid`, and `totalInterestPaid`. Notice, that we obtain the over-payment for a month from the `overpayments` dictionary, where the default over-payment (if the key doesn't exist) is 0 (`get(overpayments, month, 0)`).

Let's do a quick sanity check. Previously (Section 32.2), we said that the regular payment off `mortgage1` ($200,000 at 6.49% for 20 years) will roughly yield the principal of $200,000 and the interest of $157,593. Let's see if we get that.

```
payOffMortgage(mortgage1)
```

```
Summary(199999.99999999988, 157592.5713666817, 240)
```

OK, so finally, we are ready to answer our questions.

How much money can we potentially save in the case of `mortgage1` ($200,000, 6.49%, 20 years) (see Section 32.2) overpaid regularly every month with $200.

```
function getTotalCost(s::Summary)::Real
    return s.principal + s.interest
end

function getTotalCostDiff(m::Mortgage,
    overpayments::Dict{Int, <:Real})::Real
    s1::Summary = payOffMortgage(m)
    s2::Summary = payOffMortgage(m, overpayments)
    return getTotalCost(s1) - getTotalCost(s2)
end
```

And (here we use comprehensions[302]with a dictionary):

```
getTotalCostDiff(mortgage1,
    Dict(i => 200 for i in 1:mortgage1.numMonths)) |> fmt
```

37,444 USD

Quite a penny (see also Figure 17).

[302]https://b-lukaszuk.github.io/RJ_BS_eng/julia_language_repetition.html#sec:julia_language_comprehensions

Figure 17: Overpaying a mortgage ($200,000, 6.49%, 20 years) with $200 monthly (estimation may not be accurate).

And now let's overpay `mortgage2` ($200,000, 4.99%, 30 years) with $200 monthly.

```
getTotalCostDiff(mortgage2,
    Dict(i => 200 for i in 1:mortgage2.numMonths)) |> fmt
```

60,987 USD

The total savings appear to be even greater than for `mortgage1`, still the total cost seems to be greater for the overpaid `mortgage2` (see Figure 17 and Figure 18).

Figure 18: Overpaying a mortgage ($200,000, 4.99%, 30 years) with $200 monthly (estimation may not be accurate).

OK, time for the next question.

For `mortgage1` which one is more worth it: to overpay it every month with $200 or to overpay it only once, let's say in month 13, with $20,000?

```
(
    getTotalCostDiff(mortgage1,
        Dict(i => 200 for i in 1:mortgage1.numMonths)) |> fmt,
    getTotalCostDiff(mortgage1, Dict(13 => 20_000)) |> fmt
)
```

```
("37,444 USD", "40,972 USD")
```

Interesting, the above output indicates that we would be able to save more, with an early (month 13) over-payment of a vast sum of money ($20,000, 10% of our initial principal) than just by regularly overpaying the mortgage with small sums of it ($200, 0.1% of our initial principal).

Out of pure curiosity, let's see how much we could save when we combine the two (we overpay $200 every month, except for month 13, where we overpay $20,000)

```
customOverpayments = Dict(i => 200 for i in 1:mortgage1.numMonths)
customOverpayments[13] = 20_000
getTotalCostDiff(mortgage1, customOverpayments) |> fmt
```

65,257 USD

And now, the final question. Which one is better: to overpay `mortgage1` with $20,000 in month 13, or to put this $20,000 into a bank deposit that pays 5% yearly for the 19 years (roughly the remaining duration of the mortgage)?

```
(
    getTotalCostDiff(mortgage1, Dict(13 => 20_000)) |> fmt,
    # fn from chapter about compund interest
    getValue(20_000, 5, 19) - 20_000 |> fmt
)
```

```
("40,972 USD", "30,539 USD")
```

So if the calculations were accurate in this scenario we would have saved (in nominal money) $40,972 in interests, whereas gained extra $30,593 (to our initial $20,000) on the bank deposit (compare with Section 31.2.1). Advantage over-payment.

Diffusion

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[303](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[304].

Problem

This time your job is to write a simplified diffusion[305]simulator. It may or may not be a terminal app (mine runs in a terminal).

For that purpose create a container (let's say 40x80) and fill its left half with let's say 150 molecules (like in Figure 19) that are sped by Brownian motion[306]. See if the diffusion works, by comparing initial distribution of particles with the distribution after a few thousand cycles. To make it simpler you don't have to handle the collisions between the molecules (assume they pass each other in the third dimension), but make sure they won't go out of the container.

> ***WARNING:*** While running the program in a terminal the screen may flicker. If that's a problem (you may feel unwell) then you can skip this exercise. Remember that you should be able to abort the program (terminal application) at any time by pressing Ctrl-C.

[303]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/diffusion
[304]https://docs.julialang.org/en/v1/stdlib/Pkg/
[305]https://en.wikipedia.org/wiki/Diffusion
[306]https://en.wikipedia.org/wiki/Brownian_motion

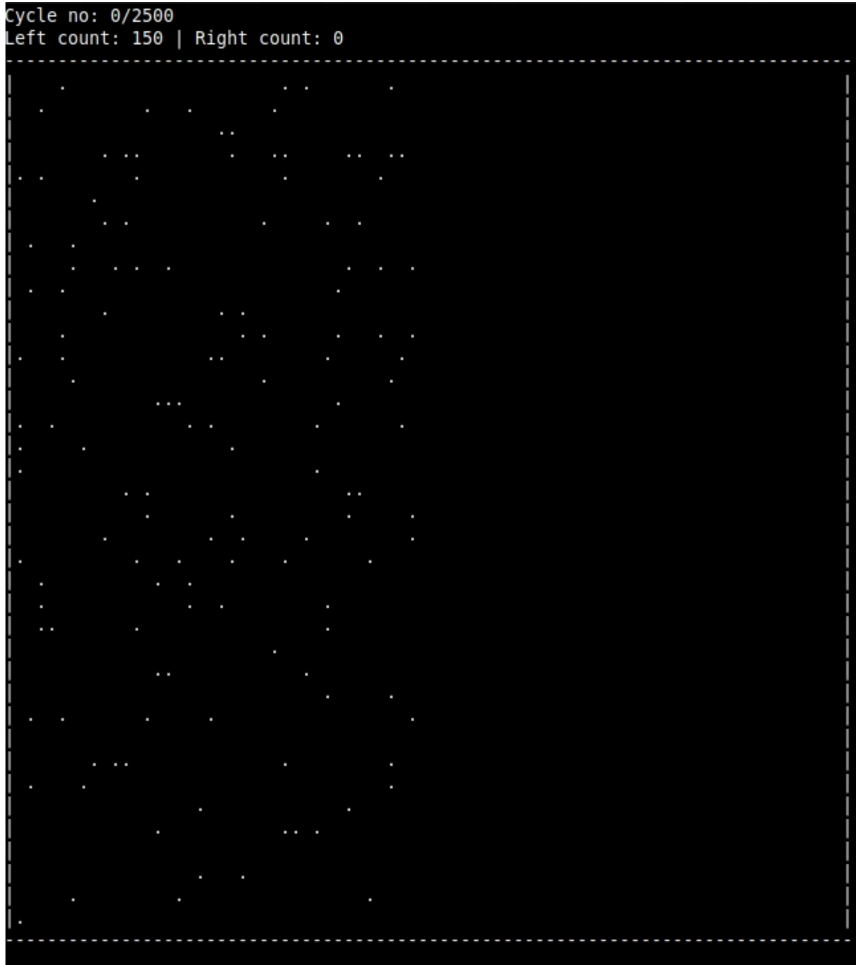


Figure 19: Simplified diffusion simulation (initial frame).

Solution

Let's start with the container and its functionality.

```
const N_COLS = 80
const N_ROWS = 40

function addBorders!(container::Matrix{Char})::Nothing
    container[:, 1] .= '|'
    container[:, N_COLS] .= '|'
    container[1, :] .= '-'
```

```
    container[N_ROWS, :] .= '-'
    return nothing
end

function getEmptyContainer()::Matrix{Char}
    container::Matrix{Char} = fill(' ', N_ROWS, N_COLS);
    addBorders!(container)
    return container
end

function printContainer(container::Matrix{Char})::Nothing
    for r in 1:N_ROWS
        println(container[r, :] |> join)
    end
    return nothing
end

function clearDisplay(nLinesUp::Int)::Nothing
    @assert 0 < nLinesUp "nLinesUp must be a positive integer"
    # "\033[xxxA" - xxx moves cursor up xxx LINES
    print("\033[" * string(nLinesUp) * "A")
    # "\033[0J" - clears from cursor position till the end of the screen
    print("\033[0J")
    return nothing
end
```

The `container` is just a `Matrix` (a table) of characters (`Chars`) that's constrained by the borders (`|` and `-`) and initially contains nothing inside (`fill(' ', N_ROWS, N_COLS)`).

Now we need our `molecules`. The above will be defined as a vector of positions denoting their locations (row and column) within the matrix (our `container`).

```
const Pos = Tuple{Int, Int} # position (row, col) in 2D container

function isWithinContainer(molecule::Pos)::Bool
    row, col = molecule
    # accounts for borders
    return (1 < row < N_ROWS) && (1 < col < N_COLS)
end
```

At first `molecules` will occupy random positions.

```
# assumption: molecules may pass through each other
# (or occupy the same pixel in 2D) since they move
# past each other in the third (not drawn) dimension
function placeMoleculesRandomly!(molecules::Vec{Pos},
                                 rowMin::Int, rowMax::Int,
                                 colMin::Int, colMax::Int)::Nothing
    @assert(isWithinContainer((rowMin, colMin)),
            "(rowMin, colMin) outside of container")
    @assert(isWithinContainer((rowMax, colMax)),
            "(rowMax, colMax) outside of container")
    r::Int, c::Int = 0, 0
    for i in eachindex(molecules)
        r = rand(rowMin:rowMax)
        c = rand(colMin:colMax)
        molecules[i] = (r, c)
    end
    return nothing
end

const MOLECULE = '.'

function addMolecules2container!(molecules::Vec{Pos},
                                 container!::Matrix{Char})::Nothing
    for molecule in molecules
        if isWithinContainer(molecule)
            container![molecule...] = MOLECULE
        end
    end
    return nothing
end
```

Using `rowMin`/`rowMax`, `colMin`/`colMax` allows to place the molecules only in some part of the matrix (per task specification it will be the left side of `container`). Notice `!` character in `addMolecules2container!`. Per Julia's convention it was added to the name of the function that modifies its contents. However, both `molecules` (`Vec{Pos}`) and `container` (`Matrix{Char}`) are passed by reference. So a question may arise which one of the two (or maybe both) will get modified. To help with the answer the second parameter was named `container!` to emphasize that only it will be modified by the function. On the other hand, `placeMoleculesRandomly!` may modify at most one of its arguments (`molecules`) so there is no need for an extra `!` which might be confusing at first glance. Anyway, the key thing is that based on the

positions (`molecules`) we placed a marker `MOLECULE = '.'` in our `container` which later on will be displayed to the user.

Time to implement Brownian motion[307] or:

> [...] a normal distribution with the mean $\mu = 0$ and variance $\sigma^2 = 2Dt$ usually called Brownian motion B_t [...]
>
> (quote from the Wiki link provided above)

Here, we are OK with all the molecules being identical (and sharing identical properties). Moreover, for simplicity we'll just assume that `D = 0.5` and `t = 1`, hence `2Dt = 1`. This last action will give us a normal distribution[308] with the mean $\mu = 0$ and variance $\sigma^2 = 1$ (standard deviation `sd` is also 1, since $sd = \sqrt{variance}$). Luckily, that is what the built-in randn[309] function provides. Hence, we will calculate the new position of a molecule to be `new_position = old_position + shift` (`randn()` provides the shift) and implement it like so:

```
round2int(f::Flt)::Int = round(Int, f)

function getNewPosition(molecule::Pos)::Pos
    rowShift::Int = randn() |> round2int
    colShift::Int = randn() |> round2int
    return molecule .+ (rowShift, colShift)
end

const N_MOLECULES = 150

# assumption: molecules may pass through each other
# (or occupy the same pixel in 2D) since they move
# past each other in the third (not drawn) dimension
function make1BrownianCycleShift!(molecules::Vec{Pos})::Nothing
    i::Int = 1
    newPos::Pos = (0, 0)
    while i <= N_MOLECULES
        newPos = getNewPosition(molecules[i])
        if isWithinContainer(newPos)
```

[307] https://en.wikipedia.org/wiki/Brownian_motion
[308] https://b-lukaszuk.github.io/RJ_BS_eng/statistics_normal_distribution.html
[309] https://docs.julialang.org/en/v1/stdlib/Random/#Base.randn

```
            molecules[i] = newPos
            i += 1
        end
    end
    return nothing
end
```

The terminal display will require to give a shift in integers, hence `round2int` implemented as single expression function syntax[310]. Moreover, we cannot allow a particle to go outside the walls of the container, thus the `if isWithinContainer(newPos), etc.` statement. Together with the surrounding `while` block it makes sure that the molecule 'falls back' to the container. Effectively, it kind of simulates a reflection of a molecule from the walls of the vessel in a random direction.

Now we are almost ready for running our simulation, but first two, rather self-explanatory, functions:

```
const N_CYCLES = 2_500

getCol((_, c)::Pos)::Int = c

function getRightLeftCountsInfo(molecules::Vec{Pos})::Str
    colsWithMolecules::Vec{Int} = map(getCol, molecules)
    midCol::Int = round2int(N_COLS/2)
    lCount::Int = sum(colsWithMolecules .<= midCol)
    rCount::Int = sum(colsWithMolecules .> midCol)
    return "Left count: $lCount | Right count: $rCount"
end

function redrawDisplay(container::Matrix{Char},
                       molecules::Vec{Pos},
                       nCycle::Int)::Nothing
    clearDisplay(N_ROWS+2) # container + 2 info lines below
    println("Cycle no: $nCycle/$N_CYCLES")
    println(getRightLeftCountsInfo(molecules))
    printContainer(container)
    return nothing
end
```

Finally, crème de la crème, the simulation itself.

[310] https://en.wikibooks.org/wiki/Introducing_Julia/Functions#Single_expression_functions

```
const DELAY_SEC = 0.1

function simulateBrownianMotions(nCycles::Int=N_CYCLES)::Nothing
    @assert 500 <= nCycles <= 1e5 "nCycles must be in range [500, 1e5]"

    container::Matrix{Char} = getEmptyContainer()
    molecules::Vec{Pos} = fill((0, 0), N_MOLECULES)
    placeMoleculesRandomly!(molecules,
                            # adjusted for borders
                            2, N_ROWS-1,
                            2, round2int((N_COLS-2)/2)
                            )
    addMolecules2container!(molecules, container)
    redrawDisplay(container, molecules, 0)

    for cycleNumber in 1:nCycles
        make1BrownianCycleShift!(molecules)
        emptyContainer!(container)
        addMolecules2container!(molecules, container)
        sleep(DELAY_SEC)
        redrawDisplay(container, molecules, cycleNumber)
    end

    return nothing
end
```

Nothing special here, we just declare `container`/`molecules` and initialize them with the appropriate values. Then, `for` each cycle `1:nCycles` we `make1BrownianCycleShift`, remove the old molecule symbols from the `container` (`emptyContainer`), add the symbols of new, shifted molecules (`addMolecules2container`), pause for a moment (`sleep`) and redraw everything (`redrawDisplay`).

All that's left to do is to add the `main` function since the app is meant to be run from terminal.

```
const SECS_PER_MIN = 60
const DURATION_SEC = DELAY_SEC * N_CYCLES
const DURATION_MIN = DURATION_SEC / SECS_PER_MIN

rnd2(x::Flt)::Flt = round(x, digits=2)

function main()::Nothing
    println("\nThis is a toy program that models simplified diffusion.")
    println("Note: your terminal must support ANSI escape codes.\n")
```

```
    println("Estimated execution time of the program:")
    print("$(rnd2(DURATION_SEC)) seconds or $(rnd2(DURATION_MIN)) ")
    println("minutes.")
    print("WARNING: the screen may flicker ")
    println("(Ctrl-C should abort the program).")

    # y(es) - default choice (also with Enter), anything else: no
    println("\nContinue with the simulation? [Y/n]")
    choice::Str = readline()
    if lowercase(strip(choice)) in ["y", "yes", ""]
        simulateBrownianMotions()
    end

    println("\nThat's all. Goodbye!")

    return nothing
end

if abspath(PROGRAM_FILE) == @__FILE__
    main()
end
```

The end result is to be seen below (Figure 20).

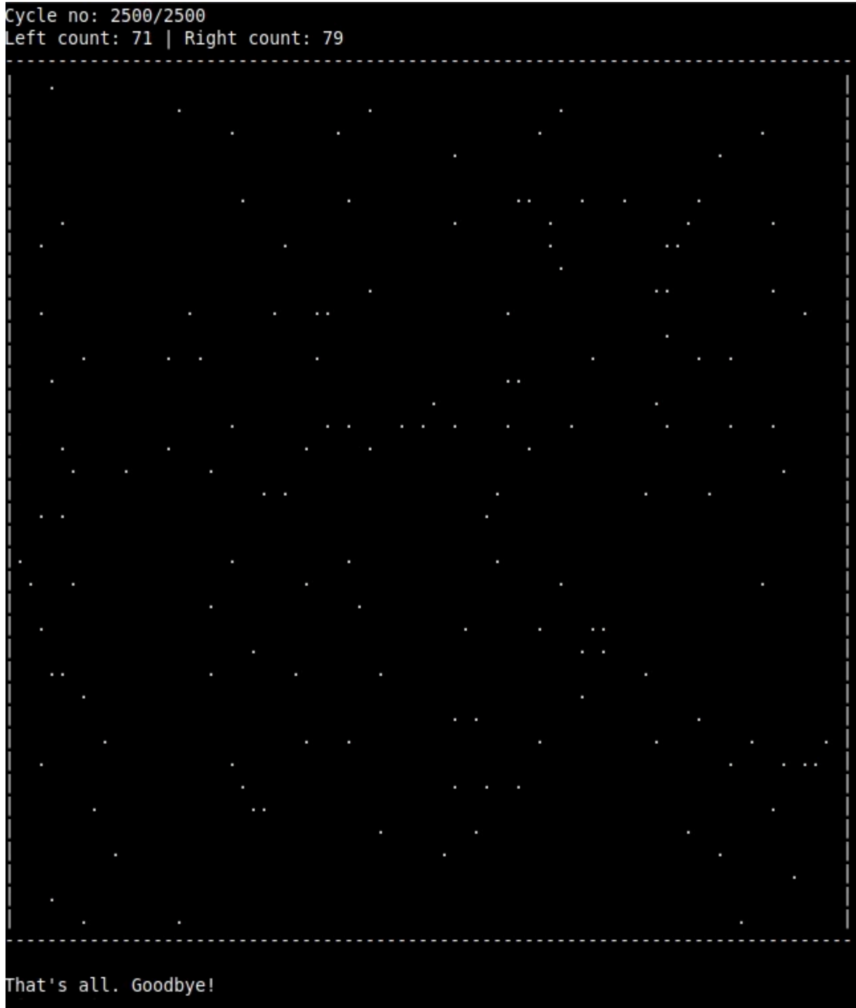


Figure 20: Simplified diffusion simulation (final frame).

Amazing. So the diffusion works in the room you find yourself in and even in our oversimplified simulation. All it took was a mock-up of Brownian motion, a reflection from a wall of the container and a few thousand cycles of random shifts to properly mix the particles.

Transcription

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[311](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[312].

Problem

The genetic material of an eucariotic cell is located in its nucleus in the form of a nucleic acid (DNA[313]to be precise). At one point DNA fragments serve as matrices to produce proteins that build our bodies and perform some actions within it (e.g. like hormones or enzymes). Such a transformation takes two steps called transcription[314]and translation[315].

During the first process, DNA's double helix is unwind and one of the strands (called template strand) is rewritten to mRNA (hence transcription) according to the table presented below.

```
 DNA | mRNA
-----+------
 'c' | 'g'
 'g' | 'c',
 'a' | 'u',
 't' | 'a'
```

[311]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/transcription

[312]https://docs.julialang.org/en/v1/stdlib/Pkg/

[313]https://en.wikipedia.org/wiki/DNA

[314]https://en.wikipedia.org/wiki/Transcription_(biology)

[315]https://en.wikipedia.org/wiki/Translation_(biology)

Here: `a`, `c`, `g`, `t`, `u` are the shortcuts (also written in uppercase) for the nucleic acids' molecular components (nucleotide bases) called `adenine`, `cytosine`, `guanine`, `thymine`, and `uracil`.

This time your task is to read the data from the file: `dna_seq_template_strand.txt` (to be found in the code snippets for this chapter[316]). The file contains a sequence of nucleotide bases of some gene. Splice its coding parts (aka. exons[317]), which encompass the molecules at positions 2424-2610 and 3397-3542. Transcribe the obtained strand to an mRNA molecule according to the complementarity rule[318]presented in the table above.

Solution

The file `dna_seq_template_strand.txt` is quite small as you can see by using your file manager or Julia:

```
#in 'code_snippets' folder use "./transcription/
dna_seq_template_strand.txt"
#in 'transcription' folder use "./dna_seq_template_strand.txt"
filePath = "./code_snippets/transcription/dna_seq_template_strand.txt"
filesize(filePath)
```

4449

Here we defined `filePath` to our file. Next, we checked its size with filesize[319]to see it is equal to 4449 bytes. This is slightly more than 4 kilobytes (KiB). Such a small file can be easily swallowed by read[320](the recommended way below) and returned as a one long `Str` (type alias for `String`).

```
dna = open(filePath) do file
    read(file, Str)
```

[316]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/transcription

[317]https://en.wikipedia.org/wiki/Exon

[318]https://en.wikipedia.org/wiki/Complementarity_(molecular_biology)#DNA_and_RNA_base_pair_complementarity

[319]https://docs.julialang.org/en/v1/base/file/#Base.filesize

[320]https://docs.julialang.org/en/v1/base/io-network/#Base.read

```
end
dna[1:75]
```

gagctccccg gatctgtaac gggaggtctc tctcgtgggt tgtgggaggt ccgaactggc\ncggtcccac

> Note. For large files you should probably read it line by line with something like `for line in eachline(file) #do sth with line# end` or use a dedicated library.

The nucleotide bases (`a`, `c`, `t`, `g`) are grouped by 10. Moreover, notice the `\n` character on the right. It is a newline[321]character that tells the computer to print the subsequent characters from the beginning of a new line. We need to splice sequence at positions 2424-2610 and 3397-3542 so let's get rid of those extra characters to make the counting easier.

```
dna = replace(dna, " " => "", "\n" => "")
dna[1:75]
```

gagctccccggatctgtaacgggaggtctctcgtgggttgtgggaggtccgaactggccggtcccacagggga

This couldn't be simpler, we just use `replace` and `itIs => shouldBe` syntax. The spaces (`" "`) are replaced with nothing (`""`, empty string) and newlines (`"\n"`) with nothing (`""`, empty string) as well. Effectively this removed them from our `dna` string.

String splicing is easily done with indexing (if we got only ASCII[322] characters) and string concatenation operator (*) like so.

```
dnaExonsOnly = dna[2424:2610] * dna[3397:3542]
dnaExonsOnly[1:75]
```

taccgggacacctacgcggaggacggggacgaccgcgacgaccgggagacccctggactgggtcggcgtcggaa

[321]https://en.wikipedia.org/wiki/Newline

[322]https://en.wikipedia.org/wiki/ASCII

All that's left to do is to transcribe to mRNA using the complementarity rule mentioned above. First, let's rewrite it to Julia's dictionary[323].

```
dna2mrna = Dict(
    'a' => 'u',
    'c' => 'g',
    'g' => 'c',
    't' => 'a'
)
```

And now the transcription itself.

```
function transcribe(nucleotideBase::Char,
    complementarityMap::Dict{Char, Char}=dna2mrna)::Char
    return get(complementarityMap, nucleotideBase, '?')
end

(
    transcribe('a'),
    transcribe('g'),
    transcribe('x')
)
```

```
('u', 'c', '?')
```

Our transcribe function takes a character (`Char`, `String` is build of individual characters) called `nucleotideBase` and a default `complementarityMap` set to `dna2mrna`. It uses `get` to return a complementary base to `nucleotideBase` (its second argument) or a default (its third argument, in this case just return `'?'`) if a match was not found.

All that's left to do is to write a `transcribe` function for the whole string (`dnaExonsOnly`).

```
function transcribe(dnaSeq::Str)::Str
    return map(transcribe, dnaSeq)
end
```

[323] https://b-lukaszuk.github.io/RJ_BS_eng/julia_language_decision_making#sec:julia_language_dictionaries

```
mRna = transcribe(dnaExonsOnly)
(
    dnaExonsOnly[1:10],
    mRna[1:10]
)
```

```
("taccgggaca"
 "auggcccugu")
```

Here a map function applies previously defined `transcribe` on every character of `dnaSeq` and glues the obtained characters into a string.

Instead of the above two functions we could have just written

```
mRna = map(base -> get(dna2mrna, base, base), dnaExonsOnly)
(
    dnaExonsOnly[1:10],
    mRna[1:10]
)
```

```
("taccgggaca"
 "auggcccugu")
```

with the same result, but I felt that the longer version was clearer.

Finally, let's just check if the transcription produced no artifacts (`'?'` defined before).

```
findfirst(base -> base == '?', mRna) |> isnothing
```

true

Everything seems to be in order.

Translation

In this chapter I used the following external libraries.

```
import Base.Iterators.takewhile as takewhile # internal library
import BenchmarkTools as Bt # external library
```

Those were used only for the chapter's extras and are not strictly necessary to solve the task.

Anyway, I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[324](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[325].

Problem

Let's start from where we left. Read the data from the file: `mrna_seq.txt` (to be found in the code snippets for this chapter[326]). It contains the mRNA sequence we obtained previously (see Section 35.2).

Your task is to translate the language of nucleic acids to the language of proteins. To do that you will operate on triplets of nucleotide bases (also called codons). You start with the `AUG` triplet (the sequence starts with it) and replace it with a proper amino acid (proteins are build of amino acids like `String`s are build of `Char`s), in this case it is methionine. Then you move to another triplet (there are no 'commas' in the genetic code) and assign the proper amino acid according to the

[324]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/translation

[325]https://docs.julialang.org/en/v1/stdlib/Pkg/

[326]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/translation

[327]https://en.wikipedia.org/wiki/DNA_and_RNA_codon_tables#Translation_table_1

standard genetic code[327]. You finish the protein synthesis once you encounter a stop codon (`UAA`, `UAG`, `UGA`).

Since rewriting the Wikipedia's table from the link above is mundane and boring, then I paste the genetic code below in Julia's dictionary format.

```
# codon - triplet of nucleotide bases, aa - amino acid
codon2aa = Dict(
    "UUU"=> "Phe", "UUC"=> "Phe", "UUA" => "Leu", "UUG" => "Leu",
    "CUU" => "Leu", "CUC" => "Leu", "CUA" => "Leu", "CUG" => "Leu",
    "AUU" => "Ile", "AUC" => "Ile", "AUA" => "Ile", "AUG" => "Met",
    "GUU" => "Val", "GUC" => "Val", "GUA" => "Val", "GUG" => "Val",
    "UCU" => "Ser", "UCC" => "Ser", "UCA" => "Ser", "UCG" => "Ser",
    "CCU" => "Pro", "CCC" => "Pro", "CCA" => "Pro", "CCG" => "Pro",
    "ACU" => "Thr", "ACC" => "Thr", "ACA" => "Thr", "ACG" => "Thr",
    "GCU" => "Ala", "GCC" => "Ala", "GCA" => "Ala", "GCG" => "Ala",
    "UAU" => "Tyr", "UAC" => "Tyr", "UAA" => "Stop", "UAG" => "Stop",
    "CAU" => "His", "CAC" => "His", "CAA" => "Gln", "CAG" => "Gln",
    "AAU" => "Asn", "AAC" => "Asn", "AAA" => "Lys", "AAG" => "Lys",
    "GAU" => "Asp", "GAC" => "Asp", "GAA" => "Glu", "GAG" => "Glu",
    "UGU" => "Cys", "UGC" => "Cys", "UGA" => "Stop", "UGG" => "Trp",
    "CGU" => "Arg", "CGC" => "Arg", "CGA" => "Arg", "CGG" => "Arg",
    "AGU" => "Ser", "AGC" => "Ser", "AGA" => "Arg", "AGG" => "Arg",
    "GGU" => "Gly", "GGC" => "Gly", "GGA" => "Gly", "GGG" => "Gly"
)
```

For convenience scientists often use one letter abbreviations of the amino acids as defined by IUPAC[328]and displayed below.

```
# aa - amino acid,
# iupac - International Union of Pure and Applied Chemistry
aa2iupac = Dict(
    "Ala" => "A", "Arg" => "R", "Asn" => "N", "Asp" => "D",
    "Cys" => "C", "Gln" => "Q", "Glu" => "E", "Gly" => "G",
    "His" => "H", "Ile" => "I", "Leu" => "L", "Lys" => "K",
    "Met" => "M", "Phe" => "F", "Pro" => "P", "Ser" => "S",
    "Thr" => "T", "Trp" => "W", "Tyr" => "Y", "Val" => "V",
    "Stop" => "Stop"
)
```

[328]https://en.wikipedia.org/wiki/International_Union_of_Pure_and_Applied_Chemistry

So as a final touch rewrite the amino acid sequence using the one-letter abbreviations above. As a result you should obtain the following sequence:

```
# AA - amino acids
expectedAAseq = "MALWMRLLPLLALLALWGPDPAAAFVNQHLCGSHLVEAL" *
    "YLVCGERGFFYTPKTRREAEDLQVGQVELGGGPGA" *
    "GSLQPLALEGSLQKRGIVEQCCTSICSLYQLENYCN"
```

Solution

Let's start as we did before by checking the file size.

```
#in 'code_snippets' folder use "./translation/mrna_seq.txt"
#in 'translation' folder use "./mrna_seq.txt"
filePath = "./code_snippets/translation/mrna_seq.txt"
filesize(filePath)
```

333

OK, it's less than 1 KiB. Let's read it and get a preview of the data.

```
mRna = open(filePath) do file
    read(file, Str)
end
mRna[1:60]
```

auggcccuguggaugcgccuccugccccugcuggcgcugcuggcccucuggggaccugac

It looks good, no spaces between the letters, no newline (`\n`) characters (we removed them previously in Section 35.2).

The only issue with `mRna` is that `codon2aa` dictionary contains the codons (keys) in uppercase letters. All we need to do is to uppercase the letters in `mRna` using Julia's function of the same name.

```
mRna = uppercase(mRna)
mRna[1:5]
```

AUGGC

OK, time to start the translation. First let's translate a triplet of nucleotide bases (aka codon) to an amino acid.

```
# translates codon/triplet to amino acid IUPAC
function getAA(codon::Str)::Str
    @assert length(codon) == 3 "codon must contain 3 nucleotide bases"
    aaAbbrev::Str = get(codon2aa, codon, "???")
    aaIUPAC::Str = get(aa2iupac, aaAbbrev, "?")
    return aaIUPAC
end
```

The function is rather simple. It accepts a codon (like `"AUG"`), next it translates a codon to an amino acid (abbreviated with 3 letters) using previously defined `codon2aa` dictionary. Then the 3-letter abbreviation is coded with a single letter recommended by IUPAC (using `aa2iupac` dictionary). If at any point no translation was found `"?"` is returned.

Now, time for translation.

```
function translate(mRnaSeq::Str)::Str
    len::Int = length(mRnaSeq)
    @assert len % 3 == 0 "the number of bases must be a multiple of 3"
    aas::Vec{Str} = fill("", Int(len/3))
    aaInd::Int = 0
    codon::Str, aa::Str = "", ""
    for i in 1:3:len
        aaInd += 1
        codon = mRnaSeq[i:(i+2)]
        aa = getAA(codon)
        if aa == "Stop"
            break
        end
        aas[aaInd] = aa
    end
    return join(aas)
end
```

We begin with some checks for the sequence. Then, we define the vector `aas` (`aas` - amino acids) holding our result. We initialize it with empty strings using `fill` function. We will assign the appropriate amino acids to `aas` based on the `aaInd` (`aaInd` - amino acid index) which we increase with every iteration (`aaInd += 1`). In `for` loop we iterate over each consecutive index with which a triple begins (`1:3:len` will return numbers as `[1, 4, 7, 10, 13, ...]`). Every

consecutive `codon` (3 nucleotic bases) is obtained from `mRNASeq` using indexing by adding +2 to the index that starts a triple (e.g. for i = 1, the three bases are at positions 1:3, for i = 4, those are 4:6, etc.). The `codon` is used to obtain the corresponding amino acid (`aa`) with `getAA`. If the `aa` is a so-called stop codon, then we immediately `break` free of the loop. Otherwise, we insert the `aa` to the appropriate place `[aaInd]` in `aas`. In the end we collapse the vector of strings into one long string with `join`. It is as if we used the string concatenation operator `*` on each element of `aas` like so `aas[1] * aas[2] * aas[3] * ...`. Notice, that e.g. `"A" * ""` or `"A" * "" * ""` is still `"A"`. This effectively gets rid of any empty `aas` elements that remained after reaching `aa == "Stop"` and `break` early.

Let's see does it work.

```
protein = translate(mRna)
protein == expectedAAseq
```

true

Congratulations! You have successfully synthesized pre-pro-insulin, a product transformed into a hormon that saved many a live. It could be only a matter of time before you achieve something greater still.

The above (`translate`) is not the only possible solution. For instance, if you are a fan of functional programming[329]paradigm you may try something like.

```
import Base.Iterators.takewhile as takewhile

function translate2(mRnaSeq::Str)::Str
    len::Int = length(mRnaSeq)
    @assert len % 3 == 0 "the number of bases is not a multiple of 3"
    ranges::Vec{UnitRange{Int}} = map(:, 1:3:len, 3:3:len)
    codons::Vec{Str} = map(r -> mRnaSeq[r], ranges)
    aas::Vec{Str} = map(getAA, codons)
    return takewhile(aa -> aa != "Stop", aas) |> join
end
```

[329]https://en.wikipedia.org/wiki/Functional_programming

We start by defining `ranges` that will help us get particular `codons` in the next step. For that purpose you take two sequences for start and end of a codon and glue them together with `:`. For instance `map(:, 1:3:9, 3:3:9)` roughly translates into `map(:, [1, 4, 7], [3, 6, 9])` which yields `[1:3, 4:6, 7:9]`, i.e. a vector of `UnitRange{Int}`. A `UnitRange{Int}` is a range composed of `Int`s separated by one unit, so by 1, like in `4:6` (`[4, 5, 6]` after expansion) mentioned above. Next, we map over those ranges and use each one of them (`r`) to get (`->`) a respective codon (`mRnaSeq[r]`). Then, we map over the `codons` to get respective amino acids (`getAA`). Finally, we move from left to right through the amiono acids (`aas`) vector and take its elements (`aa`) as long as they are not equal `"Stop"`. As the last step, we collapse the result with `join` to get one big string.

```
protein2 = translate2(mRna)
expectedAAseq == protein2
```

true

Works as expected. The functional solution (`translate2`) often has fewer lines of code (8 vs 17 for `translate2` and `translate`, respectively). It also consists of a series of consecutive logical steps, which is quite nice. However, for a beginner (or someone that doesn't know this paradigm well) it appears more enigmatic (and therefore off-putting). Moreover, in general it is expected to be a bit slower than the more imperative for loop version (`translate`). This should be evident with long sequences [try `BenchmarkTools.@benchmark translate($mRna^20)` vs `BenchmarkTools.@benchmark translate2($mRna^20)` in the REPL (type it after `julia>` prompt)].

> Note. `$` (see above) is an interpolation of global variable recommended by the package[330]. On the other hand, `^`, (again see above) replicates a string `n` times, e.g. `"ab" ^ 3 = "ababab"`. Interestingly, although `translate(mRna^20)` and `translate2(mRna^20)` receive a strand of RNA 20 times longer

[330] https://juliaci.github.io/BenchmarkTools.jl/stable/

than `mRna` they still return the same amino acid sequence as before. Test yourself and explain why. This will also help you realize why `translate2` is slower than its counterpart.

In the said case (with `mRna^20`) the difference between ≈ 27 $[\mu s]$ and ≈ 140 $[\mu s]$ (on my laptop) shouldn't be noticeable by a human (140 $[\mu s]$ is less than 1/1'000th of a second). Therefore, if the performance is acceptable you may want to go with the functional version.

Anyway, both Section 35 and Section 36 were inspired by the lecture of this ResearchGate entry[331]based on which the DNA sequence was obtained (`dna_seq_template_strand.txt` from Section 35).

[331]https://www.researchgate.net/publication/16952023_Sequence_of_human_insulin_gene

Altruism

In this chapter I used the following libraries.

```
import Random as Rnd # internal library
```

Still, once you read the problem description you may decide to do otherwise.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[332](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[333].

Problem

The following problem was inspired by the lecture of a Richard Dawkins' book and his considerations about altruism. Alas, I borrowed the book and it's been like 15-20 years since I read it, so I don't even remember its title. Shortly, if I mess things up, the fault is mine.

Anyway, there is this interesting game theory problem called the prisoner's dilemma[334]. Imagine two accomplices being interrogated by the police for the crime they committed. The investigators separate the suspects and try to convince them to testify against each other (good cop, bad cop, isn't it?):

- if both of them remain silent, there is enough evidence to sentence them for one year each
- if both of them incriminate each other, each of them faces the sentence of two years in prison
- if one of them tells on the other (but the other remains silent), then the informer is free to go and the other person serves three years sentence

[332]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/altruism
[333]https://docs.julialang.org/en/v1/stdlib/Pkg/
[334]https://en.wikipedia.org/wiki/Prisoner%27s_dilemma

Imagine, that you are one of the two. Pause for a moment and think which is the best option for you (use pure logic, not emotions, it is guaranteed that nobody will know what you did).

If you stay silent you will be punished for sure, if you betray then you may walk away free. That being said, the betrayal is often considered to be the optimal strategy.

You could say that was a no-brainer, but the problem is deeper than you may think. Imagine there are two monkeys and each got a tick[335] on their back that they cannot remove themselves. A tick decreases survival of an animal (it spreads diseases), so does the removal of the tick (less time to find food). Therefore assume that, when:

- one monkey betrays the other, the winner gets 3 survival points, the looser gets −2 points
- both monkeys cooperate they get +2 survival points each
- both monkeys refuse to help each other, each looses 1 survival point

To make it more realistic assume that the monkeys are neighbors that will interact with each other a few hundred times in their lives (but they don't know exactly how long), and get a tick just as many times. Does this new situation makes a difference, is it worth the while to be altruistic?

Let's use Julia to answer that question. To that end we assume there are 6 monkeys in the group:

- three good:
  - naive - it always cooperates
  - unforgiving - if you betray it more than three times, it will never trust you again
  - paybacker - first it cooperates, then it replays its partner's last move
- three evil:
  - unfriendly - got a bad mood at random and may betray with the probability of 60% (p = 0.6)

[335]https://en.wikipedia.org/wiki/Tick

- abusive - got a bad mood at random and may betray with the probability of 80% (p = 0.8)
- egoist - always betrays its partner

Test which monkey ends on top if every animal interacts a random number of times (let's say 50 to 300 times) with all the other animals.

Does it make a difference, if you replace the unforgiving monkey with a gullible one (it cooperates at random 90% of the times)?

> ***Note:*** You don't need to strictly adhere to the above task description, feel free to adjust it to your level/liking.

Solution

Since this is a game theory problem, then we're going to use game terminology in the solution. Ready. Let the games begin.

First, let's define all the possible values for players (i.e. monkeys) and moves (i.e. choices) by using Julia's enums[336].

```
@enum Player naive gullible unforgiving paybacker unfriendly abusive
egoist
@enum Choice cooperate=0 betray=1
```

Now, whenever we use `Player` in our code (as a variable type) we will be able to use one of the seven informative and mnemonic names (`naive gullible unforgiving paybacker unfriendly abusive egoist`). The same goes for `Choice` our players will make (`cooperate` and `betray`). Note, however, that in this last case the `Choices` are followed by the number code. We didn't have to do that since by default the enums are internally stored as consecutive integers that start at 0. Still, we did it to emphasize our plan to use this property of our new type in the near future. Namely, per problem description `unforgiving` monkey always betrays its partner when it was itself betrayed more than three times in the past. To that end we will need to count the betrayals.

[336] https://docs.julialang.org/en/v1/base/base/#Base.Enums.Enum

```
import Base: +

function +(c1::Choice, c2::Choice)::Int
    return Int(c1) + Int(c2)
end

function +(n::Int, c::Choice)::Int
    return n + Int(c)
end
```

We start by importing the `+` function from `Base` package and make the versions of it (aka methods) that know how to handle our `Choice` enum. Simply, if we add two `Choices` (`c1` and `c2`) together, we add the underlying integers (`Int(c1) + Int(c2)`) and when we add an integer (`n`) to a `Choice` then again we add the integer to the `Choice`'s integer representation (`n + Int(c)`). Thanks to this little trick, we will be able to count the total of betrayals in a vector of `Choices` with the built in `sum` function (it relies on `+`)

> ***Note:*** Do not overuse this technique. In general, you should redefine the built in `Base` functions (like `+`) only on the types that you have defined yourself.

Time to write a function that will return the Player's move. According to the problem description all it needs know to do its job correctly is the `Player`'s type and its opponents previous moves.

```
import Random as Rnd

function getMove(p::Player, opponentMoves::Vec{Choice})::Choice
    # random float in range [0.0-1.0), prob [0.0-1.0], so good enough
    prob::Flt = Rnd.rand()
    if p == naive
        return cooperate
    elseif p == unforgiving
        return sum(opponentMoves, init=0) > 3 ? betray : cooperate
    elseif p == gullible
        return prob <= 0.9 ? cooperate : betray
    elseif p == paybacker
        return isempty(opponentMoves) ? cooperate : opponentMoves[end]
    elseif p == unfriendly
        return prob <= 0.6 ? betray : cooperate
```

```
    elseif p == abusive
        return prob <= 0.8 ? betray : cooperate
    else # egoist player
        return betray
    end
end
```

The function is a bit cumbersome to type because Julia does not have a switch statement[337]known from other programming languages. If you really must have it, then consider using Match.jl[338]as a replacement. Anyway, the code is pretty simple if you are familiar with the decision making in Julia[339]. One point to notice is that here we used the `init=0` keyword argument in `sum`. This is a default value from which we start counting the total, and it makes sure that an empty vector (`opponentMoves`) returns `0` instead of an error when used with `sum`.

Time to award our players with survival points per a round (interaction) and their choices.

```
function getPts(c1::Choice, c2::Choice)::Tuple{Int, Int}
    if c1 == c2 == cooperate
        return (2, 2)
    elseif c1 > c2
        return (3, -2)
    elseif c1 < c2
        return (-2, 3)
    else # both betray
        return (-1, -1)
    end
end
```

Notice, that the `enum` defined by us (`Choice`) got a built in ordering that by default goes in an ascending order from left to right (`cooperate` < `betray` per `@enum Choice cooperate betray`) which we used to our advantage here.

[337] https://en.wikipedia.org/wiki/Switch_statement
[338] https://github.com/JuliaServices/Match.jl
[339] https://b-lukaszuk.github.io/RJ_BS_eng/julia_language_decision_making.html

Time to write a function that takes two players as an argument and runs a random number of games (50:300 interactions) between them. In the end it returns the survival points each player obtained.

```
function playRoundsGetPts(p1::Player, p2::Player)::Tuple{Int, Int}
    pts1::Int, pts2::Int = 0, 0 # total pts
    pt1::Int, pt2::Int = 0, 0 # pts per round
    mvs1::Vec{Choice}, mvs2::Vec{Choice} = [], [] # all moves
    mv1::Choice, mv2::Choice = cooperate, cooperate # move per round
    nRounds::Int = Rnd.rand(50:300)
    for _ in 1:nRounds
        mv1, mv2 = getMove(p1, mvs2), getMove(p2, mvs1)
        pt1, pt2 = getPts(mv1, mv2)
        push!(mvs1, mv1)
        push!(mvs2, mv2)
        pts1 += pt1
        pts2 += pt2
    end
    return (pts1, pts2)
end
```

We begin by defining and initializing variables to store:

1) total number of points obtained by each player (`pts1`, `pts2`)
2) the number of points obtained by the players per single interaction (`pt1`, `pt2`)
3) all the moves made by the players during their interactions (`mvs1`, `mvs2`)
4) the moves made by each player per single interaction (`mv1`, `mv2`)
5) the number of interactions (rounds) between the players (`nRounds`)

We update the above mentioned variables after every round/ interaction took place (in the `for` loop). Finally, we return the number of points obtained by each player.

Time to set things into motion and make all the players play with each other.

```
function playGame(players::Vec{Player})::Dict{Player, Int}
    playersPts::Dict{Player, Int} = Dict(p => 0 for p in players)
    alreadyPlayed::Dict{Player, Bool} = Dict()
    for player1 in players, player2 in players
        if player1 == player2 || haskey(alreadyPlayed, player2)
```

```
            continue
        end
        pts1, pts2 = playRoundsGetPts(player1, player2)
        playersPts[player1] += pts1
        playersPts[player2] += pts2
        alreadyPlayed[player1] = true
    end
    return playersPts
end
```

Again, we start by initializing the necessary variables: the result (`playersPts`) and players that already played in our game (`alreadyPlayed`). Next, we use Julia's simplified nested for loop syntax (that we met in Section 2.2) to make all players play with each other. We prevent the player playing with themselves (`player1 == player2`). We also stop the players from playing with each other two times. Without `haskey(alreadyPlayed, player2)`, e.g. `naive` would play with `egoist` twice [once as `player1` (`naive` vs `egoist`), the other time as `player2` (`egoist` vs `naive`)]. We update the points scored by each player after every pairing (`playerPts[player1] += pts1`, `playerPts[player2] += pts2`) and return them as a result (`return playersPts`).

Let's see how it works.

```
Rnd.seed!(401) # needed to make it reproducible
playGame([naive, unforgiving, paybacker, unfriendly, abusive, egoist])
```

```
Dict{Player, Int64} with 6 entries:
  unforgiving => 490
  paybacker => 464
  egoist => 323
  unfriendly => -26
  naive => 68
  abusive => 2
```

First three competitors (monkeys) are: `unforgiving` followed by `paybacker` and `egoist`. Run the simulation a couple of times (with different `seeds`) and see the results. In general the good players (monkeys) win the podium with the evil ones in 2:1 ratio.

Interestingly, if we replace the `unforgiving` with `gullible` (it cooperates at random 90% of the times) we get something entirely different.

```
Rnd.seed!(401) # needed to make it reproducible
playGame([naive, gullible, paybacker, unfriendly, abusive, egoist])
```

```
Dict{Player, Int64} with 6 entries:
  gullible => 79
  paybacker => 395
  egoist => 615
  unfriendly => 758
  naive => 36
  abusive => 326
```

The situation seems to be reversed, The evil players (monkeys) win the podium with the good ones in 2:1 ratio. So I guess: "The only thing necessary for evil to triumph in the world is that good men do nothing[340]"

[340] https://en.wikipedia.org/?title=The_only_thing_necessary_for_evil_to_triumph_in_the_world_is_that_good_men_do_nothing&redirect=no

Shift

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[341](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[342].

Problem

Imagine that one day you received a mysterious e-mail to your inbox. At first you thought it was a spam, but it just looks like a complete gibberish. Upon more detailed inspection it turned out that the text got some regularity to it. So it might be a coded message. Just in case you saved it as `trarfvf.txt` (see the code snippets[343]), since `trarfvf` was the title of the message.

Now, a simple method to code something is to use a shift cipher (see Figure 21).

[341] https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/shift
[342] https://docs.julialang.org/en/v1/stdlib/Pkg/
[343] https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/shift

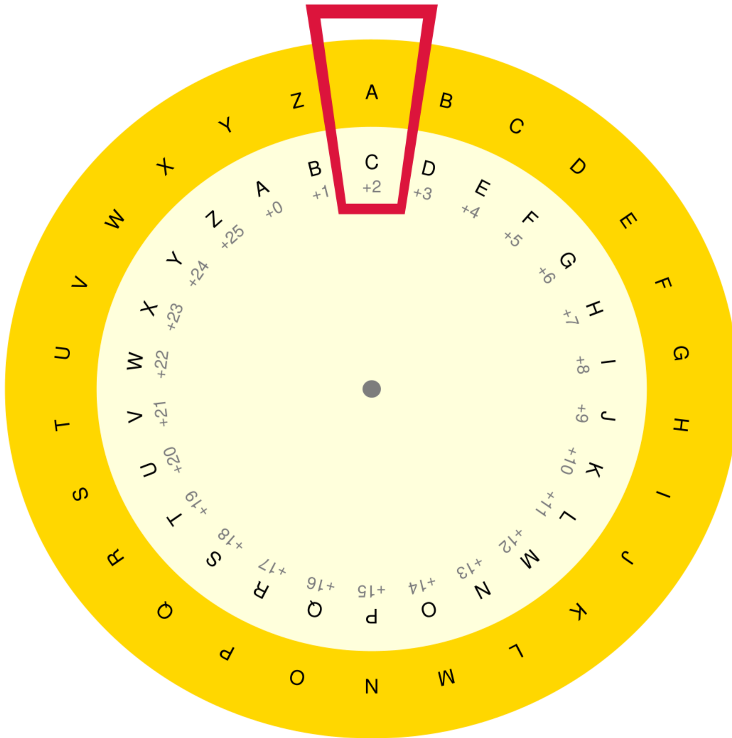


Figure 21: Coding Discs. The outer disc contains the original alphabet. The inner disc contains the alphabet shifted by 2 characters

To that end you cut two discs out of paper with all the characters from the alphabet on them. You shift the inner disk by a certain number of fields (+2 in Figure 21). In order to encode a letter you move the red tick around to that character (in the outer circle). Next you read the encoded letter in the inner circle (as pointed by the tick). If a letter or a symbol from the original text is not in the disk you just retype it as it is.

This way, coding the phrase “JULIA :)” with shift +2 from Figure 21 would give us “LWNKC :)”

Anyway, here is a task for you. Use a frequency analysis[344]to figure out the shift (rotation) used to code the message found in `trarfvf.txt` (~31 KiB).

Solution

Let's approach the problem step by step.

First let's read the file's contents (`open` and `read` were discussed in Section 35.2), `uppercase` all the characters (compare with Section 36.2) and preserve only letters from the English alphabet (`filter`).

```
# the file is roughly 31 KiB
# if necessary adjust the filePath
codedTxt = open("./code_snippets/shift/trarfvf.txt") do file
    read(file, Str)
end

codedTxt = uppercase(codedTxt)

function isUppercaseLetter(c::Char)::Bool
    return c in 'A':'Z'
end

codedTxt = filter(isUppercaseLetter, codedTxt)
first(codedTxt, 20)
```

VAGURORTVAAVATTBQPER

Time to get the letter counts and frequencies.

```
function getCounts(s::Str)::Dict{Char,Int}
    counts::Dict{Char, Int} = Dict()
    for char in s
        if haskey(counts, char)
            counts[char] = counts[char] + 1
        else
            counts[char] = 1
        end
    end
    return counts
end

function getFreqs(counts::Dict{Char, Int})::Dict{Char, Flt}
```

[344]https://en.wikipedia.org/wiki/Letter_frequency

```
    total::Int = sum(values(counts))
    return Dict(k => v/total for (k, v) in counts)
end

function getFreqs(s::Str)::Dict{Char, Flt}
    return s |> getCounts |> getFreqs
end
```

The code is rather simple. Moreover it is quite similar to `getCounts` and `getProbs` that I discussed it in detail in my previous book[345]so give it a sneak peak if you need a more thorough explanation (I apply the DRY principle here).

According to this Wikipedia's page[346]the letter that occurs most often in English is `E` (frequency: 0.127 or 12.7%, compare with this discussion[347]). Time to see which letter is the most frequent in our encoded text.

```
codedLetFreqs = getFreqs(codedTxt)
[k => v for (k, v) in codedLetFreqs if v > 0.12]
```

```
'R' => 0.13374233128834356
```

And the winner is `R`. Interestingly, in the metal insides of a computer letters are represented as numbers (see Section 14.1 and ASCII[348]). We can use this to our advantage and quickly obtain the shift.

```
'R' - 'E' # ASCII: 82 - 69
```

13

And so it turns out, that our encrypted message was coded using a shift cipher with the rotation of 13 (we will verify this finding in Section 39). If we were even more stubborn, we could display both

[345]https://b-lukaszuk.github.io/RJ_BS_eng/statistics_prob_theor_practice.html

[346]https://en.wikipedia.org/wiki/Letter_frequency

[347]https://b-lukaszuk.github.io/RJ_BS_eng/statistics_intro_probability_definition.html

[348]https://en.wikipedia.org/wiki/ASCII

the frequencies on a graph like Figure 22 (we do not expect the fit to be perfect).

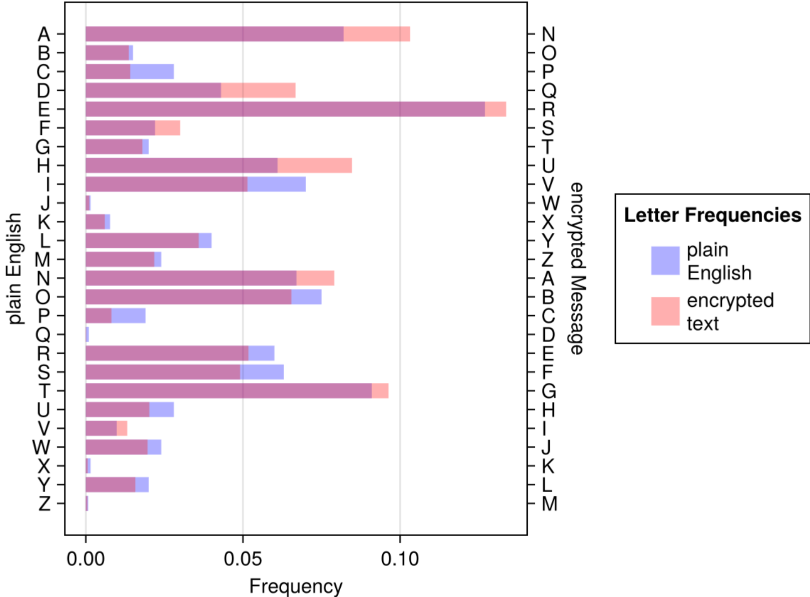


Figure 22: Frequency analysis of an encrypted text.

Caesar

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[349](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[350].

Problem

We finished the previous chapter (see Section 38.2) by stating that the file `trarfvf.txt` contains a text coded with a substitution cipher with the shift (rotation) of 13 characters. This turns out to be the Caesar cipher[351]used by the famous Roman emperor in antiquity (breaking the cipher without a computer program and sufficient amount of text is not an easy task).

So here is an exercise for you, write a computer program that can code and decode a textual message with a substitution cipher of any shift. Use it to decrypt the message contained in `trarfvf.txt` (~31 KiB). Feel free to decipher the file name (`trarfvf`) itself as well.

Solution

Let's start by writing a function that will create alphabets for the outer and inner rings of the discs from Figure 23 .

[349]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/caesar
[350]https://docs.julialang.org/en/v1/stdlib/Pkg/
[351]https://en.wikipedia.org/wiki/Caesar_cipher

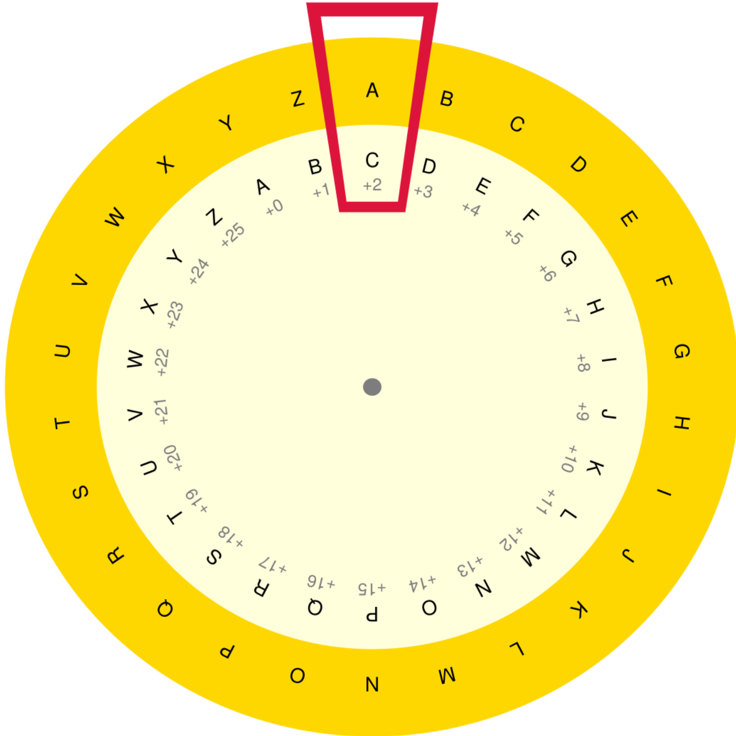


Figure 23: Coding Discs. The outer disc contains the original alphabet. The inner disc contains the alphabet shifted by 2 characters

```
function getAlphabets(rotBy::Int, upper::Bool)::Tuple{Str, Str}
    alphabet::Str = upper ? join('A':'Z') : join('a':'z')
    rot::Int = abs(rotBy) % length(alphabet)
    rotAlphabet::Str = alphabet[(rot+1):end] * alphabet[1:rot]
    return rotBy < 0 ? (rotAlphabet, alphabet) : (alphabet, rotAlphabet)
end
```

First we create an alphabet made of `'a'` to `'z'` letters with the desired casing (`upper`) using a StepRange[352](`'a':'z'` or `'A':'Z'`) that we `join`

[352]https://docs.julialang.org/en/v1/base/collections/#Base.StepRange

into a string. Next, we use modulo operator[353](`%` - returns reminder of a division) to get the desired rotation (`rot`). This allows us to gracefully handle the overflow of `rotBy` [e.g. when `rotBy` is 28 and `length(alphabet)` is 26 we get `28 % 26`i.e. 2 (full circle turn + shift by 2 fields)]. Then we create the rotated alphabet (`rotAlphabet`) starting at `(rot+1)` and appending (*) the beginning of the normal `alphabet`. Finally, if `rotBy` is negative (`rotBy < 0`, decrypting the message) we return `rotAlphabet` and `alphabet` to be used as outer and inner disc in Figure 23 , respectively. Otherwise `alphabet` lands in the outer ring and 'rotAlphabet' in the inner one (encrypting a message).

Time for a simple test.

```
Dict(i => getAlphabets(i, true) for i in -1:1:1)
```

```
Dict{Int64, Tuple{String, String}} with 3 entries:
  0 => ("ABCDEFGHIJKLMNOPQRSTUVWXYZ", "ABCDEFGHIJKLMNOPQRSTUVWXYZ")
  -1 => ("BCDEFGHIJKLMNOPQRSTUVWXYZA", "ABCDEFGHIJKLMNOPQRSTUVWXYZ")
  1 => ("ABCDEFGHIJKLMNOPQRSTUVWXYZ", "BCDEFGHIJKLMNOPQRSTUVWXYZA")
```

Appears to be working as intended. Now, even a greatest journey begins with a first step. Therefore, in order to encode any text we need to be able to encode a single character.

```
function codeChar(c::Char, rotBy::Int)::Char
    outerDisc::Str, innerDisc::Str = getAlphabets(rotBy, isuppercase(c))
    ind::Union{Int, Nothing} = findfirst(c, outerDisc)
    return isnothing(ind) ? c : innerDisc[ind]
end
```

We begin by obtaining `outerDisc` and `innerDisc` with `getAlphabets` that we just created. Next, we search for the index (`ind`) of the character to encode (`c`) in the `outerDisc`. The search may fail (e.g. no `,` in an alphabet) so `ind` can be either `nothing` (value `nothing` of type `Nothing` indicates a failure) or an integer hence the type of `ind` is `Union{Int, Nothing}` to depict just that. Finally, if the search failed

[353] https://docs.julialang.org/en/v1/base/math/#Base.rem

(`isnothing(ind)`) we just return `c` as it was, otherwise we return the encoded letter read from the inner ring (`innerDisc[ind]`). Observe.

```
(
    codeChar('a', 1),
    codeChar('A', 2),
    codeChar(',', 2)
)
```

```
('b', 'C', ',')
```

Once we know how to code a character, time to code a string.

```
# rotBy > 0 - encrypting, rotBy < 0 - decrypting, rotBy = 0 - same msg
function codeMsg(msg::Str, rotBy::Int)::Str
    coderFn(c::Char)::Char = codeChar(c, rotBy)
    return map(coderFn, msg)
end
```

And voila. Notice, that first we created `coderFn`, which is a function (single expression function[354]) inside of a function (`codeMsg`). Now we can neatly use it with `map`.

OK, time to decipher our enigmatic message (remember that in Section 38.2 we figured out that the shift is equal 13).

```
# be sure to adjust the path
# the file's size is roughly 31 KiB
codedTxt = open("./code_snippets/shift/trarfvf.txt") do file
    read(file, Str)
end

decodedTxt = codeMsg(codedTxt, -13)
"<<" * first(decodedTxt, 54) * ">>"
```

<<In the beginning God created the heaven and the earth.>>

Hmm, it looks suspiciously like the first phrase from the Bible.

[354] https://en.wikibooks.org/wiki/Introducing_Julia/Functions#Single_expression_functions

```
codeMsg("trarfvf", -13)
```

genesis

And indeed, even the file name indicates that it is (a part of) the Book of Genesis.

We could stop here or try to improve our solution a bit.

Let's try to define `getEncryptionMap` that will be an equivalent of our `getAlphabets`.

```
function getEncryptionMap(rotBy::Int)::Dict{Char, Char}
    encryptionMap::Dict{Char, Char} = Dict()
    alphabet::Str = join('a':'z')
    rot::Int = abs(rotBy) % length(alphabet)
    rotAlphabet::Str = alphabet[(rot+1):end] * alphabet[1:rot]
    if rotBy < 0
        alphabet, rotAlphabet = rotAlphabet, alphabet
    end
    for i in eachindex(alphabet)
        encryptionMap[alphabet[i]] = rotAlphabet[i]
        encryptionMap[uppercase(alphabet[i])] =
uppercase(rotAlphabet[i])
    end
    return encryptionMap
end
```

The function is pretty similar to the one previously mentioned, except for the fact that it returns a dictionary with the alphabet on the outer disc being the keys and the letters on the inner disc are its values (compare with Figure 23). Moreover, the map contains both lower- and upper-case characters.

Time to code a character.

```
function code(c::Char, encryptionMap::Dict{Char, Char})::Char
    return get(encryptionMap, c, c)
end
```

To that end we used the built in `get` function that looks for a character (`c`, `get`'s second argument) in `encryptionMap`, if the search failed it returns the character as a default value (`c`, `get`'s third argument).

Finally, let's code the message.

```
function code(msg::Str, rotBy::Int)::Str
    encryptionMap::Dict{Char, Char} = getEncryptionMap(rotBy)
    coderFn(c::Char)::Char = code(c, encryptionMap)
    return map(coderFn, msg)
end
```

```
code (generic function with 2 methods)
```

Notice, that we defined two different versions (aka methods) of `code` function. Julia will choose the right one during the invocation based on the type of the arguments.

Time for a test.

```
codeMsg(codedTxt, -13) == code(codedTxt, -13)
```

true

Works the same, still in the above case `codeMsg(codedTxt, -13)` takes tens of milliseconds to execute, whereas `code(codedTxt, -13)` only hundreds of microseconds (on my laptop). The human may not tell the difference, but we obtained some 50x speedup thanks to the faster lookups in dictionaries (sometimes called hash maps in other programming languages) and the fact that we do not generate our discs anew for every letter we code.

Vigenere

In this chapter I used the following libraries.

```
import CairoMakie as Cmk # external library
```

Still, once you read the problem description you may decide to do otherwise.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[355](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[356].

Problem

Let's continue on the topic of ciphers. This time your task is to read the Wikipedia's description on the Vigenère cipher[357]and write a program that allows for coding and decoding of messages with it. Use the Wikipedia's minimal test to make sure it works correctly, i.e. "attacking tonight" coded with the key "oculorhinolaryngology" should return "ovnlqbpvt hznzeuz".

Once you got it, use it to code the text in `genesis.txt` (to be found here[358]) with a passphrase: "Julia rocks, believe in its magic." and compare the letters distribution in the text before and after coding. Pause for a moment and think can the new cipher be easily broken by a frequency analysis we used in Section 38 .

Solution

OK, so the technique relies on applying different Caesar cipher to every letter of the plain text. The shift used by the Caesar cipher is

[355]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/vigenere
[356]https://docs.julialang.org/en/v1/stdlib/Pkg/
[357]https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher
[358]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/vigenere

established by using the provided passphrase. Therefore, we can safely reuse some of the code from our previous solution (see Section 39.2).

```
function getAlphabets(rotBy::Int, upper::Bool)::Tuple{Str, Str}
    alphabet::Str = upper ? join('A':'Z') : join('a':'z')
    rot::Int = abs(rotBy) % length(alphabet)
    rotAlphabet::Str = alphabet[(rot+1):end] * alphabet[1:rot]
    return rotBy < 0 ? (rotAlphabet, alphabet) : (alphabet, rotAlphabet)
end

function codeChar(c::Char, rotBy::Int)::Char
    outerDisc::Str, innerDisc::Str = getAlphabets(rotBy, isuppercase(c))
    ind::Union{Int, Nothing} = findfirst(c, outerDisc)
    return isnothing(ind) ? c : innerDisc[ind]
end
```

Time to create a function that will (de)code the whole message.

```
function isAsciiLetter(c::Char)::Bool
    return isascii(c) && isletter(c)
end

function codeMsg(msg::Str, passphrase::Str, decode::Bool=false)::Str
    @assert isascii(msg) "msg must contain only ASCII characters"
    @assert(isascii(passphrase),
            "passphrase must contain only ASCII characters")
    pass::Str = filter(isAsciiLetter, lowercase(passphrase))
    pwr::Int = ceil(length(msg) / length(pass))
    pass = pass ^ pwr
    shifts::Vec{Int} = [c - 'a' for c in pass]
    if decode
        shifts .*= -1
    end
    result::Vec{Char} = Vec{Char}(undef, length(msg))
    shiftsInd::Int = 1
    for (ind, char) in enumerate(msg)
        result[ind] = codeChar(char, shifts[shiftsInd])
        if isAsciiLetter(char)
            shiftsInd += 1
        end
    end
    return result |> join
end
```

We begin with come preparatory code. First, we `lowercase` the `passphrase` and retain only ‘a’:‘z’ characters (`isAsciiLetter`). Next, we check how many times longer is the coded message (`length(msg)`)

in comparison to the passphrase (`length(pass)`). The `ceil` function rounds the obtained float to the nearest whole number higher than or equal to it. Finally, we repeat `pass` the necessary number of times (`^ pwr`) to make sure it is at least as long as our `msg`. Next, we obtain the shifts by subtracting `'a'` from every character (`c`) in the passphrase (`pass`). For decoding (`if decode`) purposes we change the signs in `shifts` to the opposite (`.*= -1`, where `.*` multiplies every shift by `-1` and `=` re-assigns them to the `shifts` variable). We initialize an empty vector of chars of a given length (`Vec{Char}(undef, length(msg))`). Afterwards, we traverse (`for`) all characters (`char`) and their indices (`ind`) in `msg`. We code a `char` with the proper shift (`shiftsInd`). If the coded character was a letter (`isAsciiLetter(char)`) we update the `shiftsInd` (we use the next shift for coding of another letter). Finally, we concatenate the chars (`result`) into a string (`join`) that we return.

Time for our minimal test.

```
(
    codeMsg("attacking tonight", "oculorhinolaryngology"),
    codeMsg("Attacking tonight", "oculorhinolaryngology"),
    codeMsg("attacking Tonight", "oCulorhinolaryngoloGY")
)
```

```
("ovnlqbpvt hznzeuz", "Ovnlqbpvt hznzeuz", "ovnlqbpvt Hznzeuz")
```

Yep, we got the expected "ovnlqbpvt hznzeuz" (with casing consistent with the original message).

And now for the second part of this exercise. We will code `genesis.txt` with the passphrase "Julia rocks, believe in its magic." and see if it changed the frequency distribution of the letters (no single, clearly dominant letter).

To this end we will mostly use the code from Section 38.2 so go there if you need detailed explanations.

First, lets read the text in and encode it.

```
# adjust the file path if necessary
# genesis.txt ~31 KiB
plainTxt = open("./code_snippets/vigenere/genesis.txt") do file
    read(file, Str)
end
passphrase = "Julia rocks, believe in its magic."
codedTxt = codeMsg(plainTxt, passphrase)

(
    first(plainTxt, 16),
    first(codedTxt, 16),
)
```

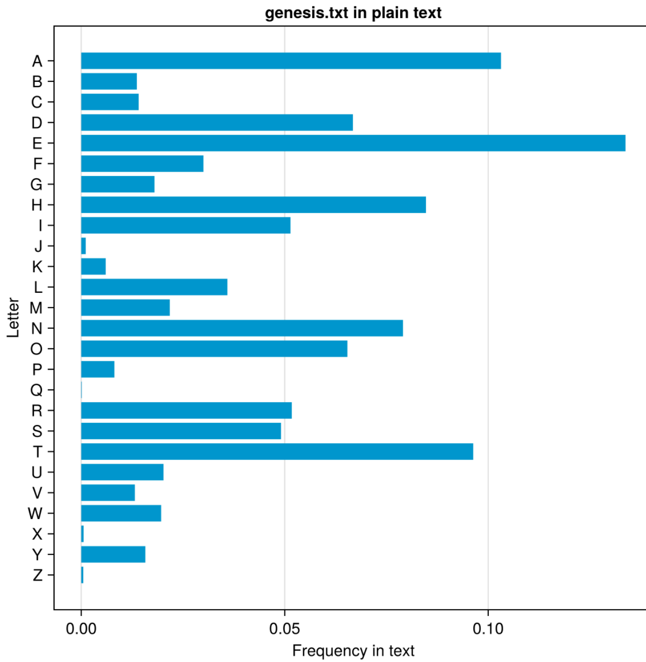
```
("In the beginning", "Rh epe ssisfomyo")
```

Now let's use the functions to obtain the letter frequencies.

```
plainTxt = filter(isAsciiLetter, uppercase(plainTxt))
codedTxt = filter(isAsciiLetter, uppercase(codedTxt))

function getCounts(s::Str)::Dict{Char,Int}
    counts::Dict{Char, Int} = Dict()
    for char in s
        if haskey(counts, char)
            counts[char] = counts[char] + 1
        else
            counts[char] = 1
        end
    end
    return counts
end

function getFreqs(counts::Dict{Char, Int})::Dict{Char,Flt}
    total::Int = sum(values(counts))
    return Dict(k => v/total for (k, v) in counts)
end

function getFreqs(s::Str)::Dict{Char,Flt}
    return s |> getCounts |> getFreqs
end
```

And finally, cherry on the cake. Instead of just printing the percentages, we will draw the comparison of the letters distribution in both texts.

```
import CairoMakie as Cmk

function drawFreqComparison(txt1::Str, title1::Str,
                            txt2::Str, title2::Str)::Cmk.Figure
    letFreqs1::Dict{Char, Flt} = getFreqs(txt1)
    letFreqs2::Dict{Char, Flt} = getFreqs(txt2)
    alphabet::Str = join('A':'Z')
    revAlphabet::Str = alphabet[end:-1:1]
    len::Int = length(alphabet)
    freqs1::Vec{Flt} = [get(letFreqs1, c, 0) for c in alphabet]
    freqs2::Vec{Flt} = [get(letFreqs2, c, 0) for c in alphabet]
    fig::Cmk.Figure = Cmk.Figure(size=(600, 1200))
    ax1::Cmk.Axis = Cmk.Axis(fig[1, 1], title=title1,
                             xlabel="Frequency in text",
ylabel="Letter",
                             yticks=(1:len, split(revAlphabet, "")),
                             ygridvisible=false)
    ax2::Cmk.Axis = Cmk.Axis(fig[2, 1], title=title2,
                             xlabel="Frequency in text",
ylabel="Letter",
                             yticks=(1:len, split(revAlphabet, "")),
                             ygridvisible=false)
    Cmk.linkyaxes!(ax1, ax2)
    Cmk.linkxaxes!(ax1, ax2)
    Cmk.barplot!(ax1, len:-1:1, freqs1, color=:deepskyblue3,
direction=:x)
    Cmk.barplot!(ax2, len:-1:1, freqs2, color=:deepskyblue3,
direction=:x)
    return fig
end

drawFreqComparison(plainTxt, "genesis.txt in plain text",
                   codedTxt, "genesis.txt coded with a Vigenere cipher")
```



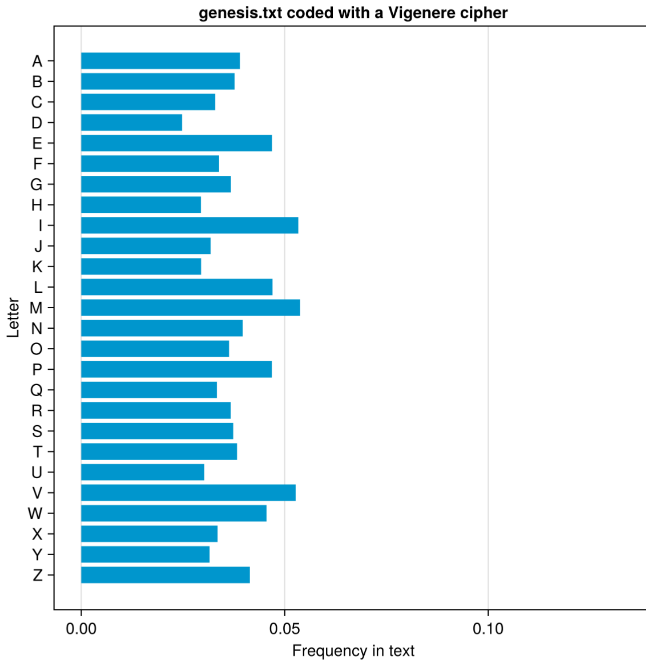


Figure 24: Frequency analysis of plain and encrypted book of Genesis.

The letter distribution definetely got more uniform[359]. This should make cracking the cipher more difficult, but still possible[360].

The performance of our `codeMsg` should be comparable to its counterpart in Section 39.2 . If you expect to work on long text messages you may improve it, e.g. by generating `encryptionMaps`. It would be a `Dict{Int, Dict{Char, Char}}` where the key (`Int`) is a possible shift (`0` to `25`) and the value (`Dict{Char, Char}`) is an `encryptionMap` (like the one produced by `getEncryptionMap` in Section 39.2) for that shift. You could generate the `encryptionMaps` only once and use them inside `codeMsg` based on the shift from `shifts::Vec{Int}`. I'll leave this as an extra exercise for you.

[359] https://en.wikipedia.org/wiki/Discrete_uniform_distribution

[360] https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher#Cryptanalysis

Game of Life

In this chapter I did not use any external libraries. Still, once you read the problem description you may decide to do otherwise. In that case don't let me stop you.

I recommend you try to solve the task on your own first. Once you finish you may compare your solution with the one in this chapter (with explanations) or with the code snippets[361](without explanations).

A reminder of how to deal with packages and *.toml files can be found here[362].

Problem

Let's finish with another classic. This time your job is to implement Conway's Game of Life[363]with a finite two dimensional grid called the universe. Each cell on the grid got some initial probability of being alive. Per the Wikipedia's description the next state of the universe is calculated as follows:

1. Any live cell with fewer than two live neighbours dies, as if by underpopulation.
2. Any live cell with two or three live neighbours lives on to the next generation.
3. Any live cell with more than three live neighbours dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

The game may look something like Figure 25 .

[361]https://github.com/b-lukaszuk/J_PRPD_eng/tree/main/code_snippets/game_of_life

[362]https://docs.julialang.org/en/v1/stdlib/Pkg/

[363]https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

```
Generation: 20/50, population: 575

................0000..............000000............................0.....0....
00............00....000...........0.......00......................00.0.0.00...
0.............00.0......0..........00......00.................000.0.0....0...00..
0..000...0..........0.0.0...........0000......................0..000.000...00...
0.000...000..000..............................................00000000....0....
0000......00...0................000....000..........................00.........
.00.....0.00000...............00..00................................0.0........
...........00..................00...............................0.00.0.0.......
................................0...00..........................0.00.0..........
..................0.............0....0.........0.0.............00.0....00.....
.................000........00.00.0..0...........00...00...........0....00.....
................00000.......0.0.0...00........000.00..00.......................
...............00...00.......00..000.........00.0..............................
..............000...000........000....0.....0...00.............0...............
...0...............00...................0.....0..................0...............
...0......................0..........00......0.0................000..000.......
...0....................00.00..........................................00.0..0.......
................000.0.....0..00......00......000.......................0.......
...............0..0.00....00.00.........000....0.......00.00.....0....0........
.............000....0..........0........00.0..00.0....0..00.....0...0..........
......0...........00......00...00.........00..0.0......0...0.......0...........
.......0.........00.......0.0..000....0..0.0..0.00.00..0..0....................
..00...00........0.........000..00........0.......0..0...0.0....0....0...0.....
..0.0.000.........0...0.....0..00..........0.0...0..00............0..0...0.....
..0....0...0....0.0.00.00....000.......0........0000..............000..........
...0000....0..00.....0..0...00......00..0.....0............0.0.000.0...0.......
...00......00............0..........0..00..0...000............0..000...0.0.0.....
....................000.....00...000.0.0.....00...........0..00.........0......
...................0.....00..0...000.000..................0..0.0..0............
..................0..00.000.00...00....00..................00..0...............
.....................0..0...00....00....00........0..................0.........
..................0.......000......00....0...000..0.0..............0..0.........
..................0...0...00..........0.......000..00.0...........0..000.....000.
....................00.0.00...............00.....00..0.........000.........000.
.....................0....0...0............0...0.0.0...0.........0...0........0.0.
......................0...0...00...........0.....0.....00...00...0000............
......000...............00.......0.........0.....0..00.00.00.0000.000.........0...
...........................00.0.........................0...0..000...........000...
...........................0...............000...........0.0................00...
...........................00..............................0....................
```

Figure 25: A frame from the Conway’s Game of Life.

WARNING: While running the program in terminal the screen may flicker. If that’s a problem (you may feel unwell) then you can skip this exercise. Remember that you should be able to abort the program (terminal application) at any time by pressing Ctrl-C.

Solution

Let’s start with our universe.

```
const N_COLS = 80
const N_ROWS = 40
const PROB_ALIVE = 0.25

const Universe = Matrix{Bool}

function getEmptyUniverse()::Universe
    return zeros(Bool, N_ROWS, N_COLS)
end

function getRandUniverse()::Universe
    # rand gives val [0-1) and not [0-1] like prob, but it will do
    return rand(Float64, N_ROWS, N_COLS) .<= PROB_ALIVE
end
```

For that we defined a few `constants` and functions. The `Universe` data type, is a synonym for a `Matrix` (`N_ROWSxN_COLS`) of `Bools`. This is a natural choice since each cell can be either alive (with the probability of `0.25`) or dead.

Next, time for printing.

```
const ALIVE_SYMBOL = 'O'
const DEAD_SYMBOL = '.'
const N_GENERATIONS = 50

function getFieldSymbol(field::Bool)::Char
    return field ? ALIVE_SYMBOL : DEAD_SYMBOL
end

function printUniverse(universe::Universe, nGeneration::Int)::Nothing
    population::Int = sum(universe)
    print("Generation: $nGeneration/$N_GENERATIONS, ")
    println("population: $population\n")
    for r in 1:N_ROWS
        println(map(getFieldSymbol, universe[r, :]) |> join)
    end
    return nothing
end

# https://en.wikipedia.org/wiki/ANSI_escape_code
function clearDisplay(nLinesUp::Int)::Nothing
    @assert 0 < nLinesUp "nLinesUp must be a positive integer"
    # "\033[xxxA" - xxx moves cursor up xxx LINES
    print("\033[" * string(nLinesUp) * "A")
    # "\033[0J" - clears from cursor position till the end of the screen
    print("\033[0J")
```

```
    return nothing
end

function reprintUniverse(universe::Universe, nGeneration::Int)::Nothing
    clearDisplay(N_ROWS+2) # +2 cause info line and newline
    printUniverse(universe, nGeneration)
    return nothing
end
```

The above (printing and reprinting) is basically a modified code from Section 34.2 . Of note, here we do not draw the borders, instead we use dead (`DEAD_SYMBOL`) and live (`ALIVE_SYMBOL`) cells.

Time to determine the next state of our `universe`. But first we need to know the number of a cell's neighbors that are alive.

```
function isCellWithinRange(row::Int, col::Int)::Bool
    return (1 <= row <= N_ROWS) && (1 <= col <= N_COLS)
end

function getNumLiveNeighbors(universe::Universe, row::Int,
col::Int)::Int
    if !isCellWithinRange(row, col)
        return 0
    end
    nAlive::Int = 0
    neighborCol::Int, neighborRow::Int = 0, 0
    for c in -1:1, r in -1:1
        neighborRow, neighborCol = row+r, col+c
        if !isCellWithinRange(neighborRow, neighborCol)
            continue
        end
        if (neighborRow == row && neighborCol == col)
            continue
        end
        if universe[neighborRow, neighborCol]
            nAlive += 1
        end
    end
    return nAlive
end
```

We assign this task to `getNumLiveNeighbors` that accepts our `universe` and the cell of interest coordinates (`row` and `col`) as its parameters. The neighbors of a cell are located in a row below, the same row as the cell, and a row above the cell's own row (`r in -1:1`).

Similarly, we look at a column to the left, the same column as the cell, and a column to the right of the cell's own column (`c in -1:1`). Hence, a neighbor's coordinates are calculated as `neighborRow = row+r` (`row` is the cell's own row, `r` is the row shift) and `neighborCol = col+c` (`col` is the cell's own column, `c` is the column shift). We examine all the possible neighbor locations with the `for` loop. If the coordinates fall outside the grid (`!isCellWithinRange` - the cell does not exist in our `universe`) then we `continue` to the next iteration (we examine next coordinates). The same goes for examining the coordinates of the cell itself (since both `r` and `c` may be equal 0). Otherwise, if a neighbor is alive (`if universe[neighborRow, neighborCol]`) we add 1 to the count (`nAlive += 1`), which we eventually `return` from the function.

Now we are ready to calculate the next state of our `universe`.

```
function shouldCellBeAlive(universe::Universe, row::Int, col::Int)::Bool
    nLiveNeighbors::Int = getNumLiveNeighbors(universe, row, col)
    if universe[row, col] && nLiveNeighbors in 2:3
        return true
    end
    return nLiveNeighbors == 3
end

function getUniverseNextState(universe::Universe)::Universe
    newUniverse::Universe = getEmptyUniverse()
    for c in 1:N_COLS, r in 1:N_ROWS
        newUniverse[r, c] = shouldCellBeAlive(universe, r, c)
    end
    return newUniverse
end
```

We start by figuring out if a cell should be alive in the next turn (`shouldCellBeAlive`). Per task specification, if a cell was previously alive (`if universe[row, col]`) and it got 2 or 3 live neighbors (`nLiveNeighbors in 2:3`) then it should be alive (`return true`). Otherwise, it should live if it was previously dead and got exactly three live neighbors (`nLiveNeighbors == 3`).

All that's left to do is to `getUniverseNextState` by examining each cell (`r` and `c`) in the `universe` and deciding its fate in the next turn (`shouldCellBeAlive(universe, r, c)`).

And now for the final touch.

```
const DELAY_SEC = 0.5

# early stop
function areAllCellsDead(universe::Universe)::Bool
    return sum(universe) == 0
end

function runGameOfLife()
    universe::Universe = getRandUniverse()
    printUniverse(universe, 0)
    for nGeneration in 1:N_GENERATIONS
        universe = getUniverseNextState(universe)
        reprintUniverse(universe, nGeneration)
        sleep(DELAY_SEC)
        if areAllCellsDead(universe)
            println("All cells are dead.")
            break
        end
    end
end
```

Let the games begin (type `runGameOfLife()` and see what happens).

The end

And so you've reached the end of this book.

The book contains forty programming exercises with reasonable (given that I'm an amateur programmer) exemplary solutions. By reasonable I mean (or rather hope for): fairly clearly written, sufficiently performant and satisfactorily robust. Still, the solutions may contain some better and worse codding ideas and are likely to depend on the day they were written on. If you put your mind at it, I'm sure you could do a better job (or break my code). With that being said, if you found a serious flaw or a bug, then feel free to open an issue[364]or a pull request[365]. Who knows, maybe I'll be able to fix them (no promises though).

Hopefully the book helped you grow as a Julia programmer as it helped to grow me. Thanks for sharing this trip.

Take care.

Bartłomiej Łukaszuk - author

[364]https://github.com/b-lukaszuk/J_PRPD_eng/issues
[365]https://github.com/b-lukaszuk/J_PRPD_eng/pulls